Software Engineering

You Wu - youwuyou@ethz.ch. version: 14/02/2024

1. Specifications and Modelling

How to document?

- **Comments:** simple, flexible; targets humans
- **Metadata:** annotations allow one to attach additional information
  1. Static processing: type checks, compiler warnings
  2. Dynamics processing: dependency injection, role-based access checks

```
@NonNull Bitmap getImage(){
  ...
}
```

```
@deprecated("no longer necessary", "1.3.1") def
removeCopies(){
  ...
}
```

- **Types and modifiers:** document basic properties; eg. memory location, inputs, results, invariants
- **Effect systems:** extensions of type systems, describe computational effects
  - exceptions, I/O effects, (de)-allocation, locking, termination, determinism

```
fun sqr: (int)->total int // mathematical total function
fun divide : (int, int)-> exn int // may raise an exception (partial)
fun turing: (tape) -> div int // may not terminate (diverge)
fun print: (string) -> console() // may write to the console
fun rand: () -> ndet int // non-deterministic
```

- **Contracts:** stylized assertions; part of the specification, not of the implementation
  - only pure expressions that do not modify the program state can be used in pre/postconditions & invariants
    - ✗ `x++`
    - ☑ `x>0, x < std::numeric_limits<int>::max()`
  - use getters and give complete range constraints
    - ✗ `0 <= self.x_ <= self.n_`
    - ☑ `0 <= GetX() < GetN()`

preconditions	The conditions the caller of a function has to fulfill to be able to perform the call. E.g., to call the `sqrt(x)` function, which computes the square root of its argument, the caller has to provide a value `x >= 0`.

postconditions	The guarantees the caller gets after the method is executed.
	1. What members don't change
	`// post-condition: GetY() == old(GetY())`
	`// ...: old(GetN()) == GetN()`
	2. What members change [case distinction]
	`// old(GetX()) == 0 => GetX() == GetN() - 1`
	`// old(GetX()) > 0  => GetX() == old(GetX()) - 1`

invariants	The conditions that all the instances of a class have to satisfy while they can be observed by the clients.
	• Thus class invariants may be temporarily violated when the class instances are not observable by clients
	In `boost::contract`, Class invariants are defined in a special `void invariant() const` member function that must be public.
	`void invariant() const {`
	`BOOST_CONTRACT_ASSERT(buffer_ != nullptr);`
	`BOOST_CONTRACT_ASSERT(0 <= size_);`
	`BOOST_CONTRACT_ASSERT(size_ <= capacity_);`
	`BOOST_CONTRACT_ASSERT(0 < capacity_);`
	`}`

What to document?

1. Member functions and constructors
   - arguments and input state
   - results and output state
   - effects/throws
2. Data structure
   - value and structural invariants
   - one-state and temporal invariants
3. Algorithms
   - behavior of code snippets (analogous to member functions)
   - explanation of control flow
   - justification of assumptions

For clients: Document the interface → How to use the code? How to call a function correctly? How the call affects the program states?

- The client interface of a class consists of constructors, public member functions & variables, external functions like << or `std::hash`
- Explicit parameters (eg. must be non-`nullptr`, point to the start? Range must be of a certain size?)
- Implicit parameters (this-object).

For implementers: Document the implementation. → How does the code work?

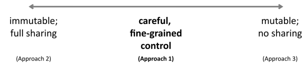
2. Modularity

General design goal: Low Coupling!

- Coupling refers to the degree of interdependence between software modules
- Tightly-coupled modules cannot be used in isolation, which makes developing, testing, changing, understanding, and reusing more difficult

1. Data coupling (Modules coupled via shared data structures)

- Problems caused by: changes in data structure; unexpected side effects; concurrency
  - Approach 1. Restricting access to data
    - ⚠ pointer access to member variable may allow capturing and leaking!
    - values shall not be changed by client code (Concurrency, unexpected side effect)
    - internal representation can be changed in the future
  - Approach 2. Making shared data immutable (Flyweight)
    - avoid unexpected side effects caused by other components changing data
    - avoid thread synchronization issues by several components changing data at same time
    - avoid invariants being broken by other components
  - Approach 3. Avoiding shared data (Pipe-and-filter)



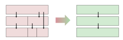
2. Procedural coupling (Modules coupled through calls)

- Problems: Callers cannot be reused without callee modules, any change in the callees may require changes in the caller
- Approach 1. Refactor code (or even duplicate functionality)

```
// Dependencies between Controller and LogEntry
class LogEntry{... bool is_error(){...}};

class Sensor{
  List<LogEntry> log_data;
  List<LogEntry>& log(){return log_data;}
};

class Controller{
  Sensor sensor;
  bool selftest(){
    auto log = sensor.log();
    for(auto& e:log)
    {if (e.is_error()) return false;}
    return true;
  }
};
```

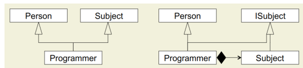
```
// After refactoring - dependencies removed
class LogEntry{... bool is_error(){...}};

class Sensor{
  List<LogEntry> log_data;
  List<LogEntry>& log(){return log_data;}

  // moved from controller to sensor
  bool no_error(){
    for(auto& e: this->log())
    {if (e.is_error()) return false;}
    return true;
  }
};

class Controller{
  Sensor sensor;
  bool selftest(){ return sensor.no_error();}
};
```

- Approach 2. Event-based communication (Observer Pattern)
- Approach 3. Restricting calls enforce policy restricting which modules a module may call
  - *Example: Multilayered/multitier architectures*
    1. A layer depends only on lower layers, has no knowledge of higher layers
    2. Layers can be exchanged



3. Class coupling (Coupled through member types, inheritance & object creation)

- Approach 1. Abstract over concrete class types
  - Use interfaces: Replace occurrences of class names by supertypes; Use the most general supertype (eg. iterators instead of vectors); Make sure data structures can be changed without affecting the code; Use templates, generics
- Approach 2. Refactor inheritance with subtyping + aggregation +delegation
  - Problem 1. Fragile base class problem: Changes in superclasses may break subclasses
  - Problem 2. Limits options for other inheritance relations. May cause conflicts with multiple inheritance; Multiple inheritance not always available (e.g. Java)



- Approach 3. Externalize object creation
  - Goal: Avoid class dependencies via object creation via
    1. Factory pattern
    2. Dependency injection
    3. Constructor parameters

```
class SymbolTable{
  Map<Ident, Type> types;

  SymbolTable(){ types =
    new TreeMap<Ident, Type>();}
}
```

```
class SymbolTable{
  Map<Ident, Type> types;

  SymbolTable(Map<Ident, Type> t){ types =
t;}
}
```

3. Design Patterns

Overview: Design patterns

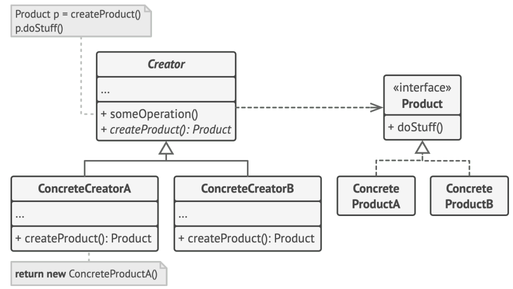
1. Creational Pattern (factory, static factory, singleton)
2. Structural Pattern (facade, flyweight, decorator)
3. Architectural Pattern (pipes and filters, model-view-controller)
4. Behavioral Pattern (observer, visitor, strategy, template)

3.1. Creational Pattern (Object Creation)

3.1.1. Factory Method

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Let a class defer instantiation to subclasses
- *Factory Method is a specialization of Template Method.*

Scheme



3.1.2. Static Factory Method

- A class provides a static method dedicated to instance creation.

Example:

```
class Response {
 public:
  static std::shared_ptr<Response> NotFoundResponse()
  { return std::make_shared<NotFoundResponse>();}

  static std::shared_ptr<Response> MarkdownResponse(const std::string& body)
  { return std::make_shared<MarkdownResponse>(body); }

  static std::shared_ptr<Response> XMLFileResponse(std::string path)
  { return std::make_shared<XMLFileResponse>(path); }
};
```

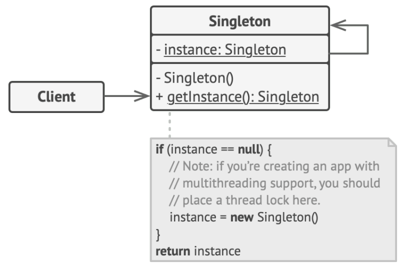
3.1.3. Singleton

Ensures a class only has one instance that gets created and provides a global point of access to it (not possible with a regular constructor)

Caution: The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.

Properties:

1. The default constructor is private
2. The static creation method `getInstance()` acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object. (Lazy initialization)



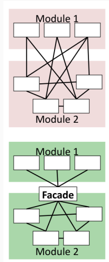
3.2. Structural Pattern (Compositional Structure)

3.2.1. Facade Pattern

- Provide a unified interface to a set of interfaces in a subsystem.
- Defines a higher-level interface that makes the subsystem easier to use
- Aggregate the selectively expose functionality

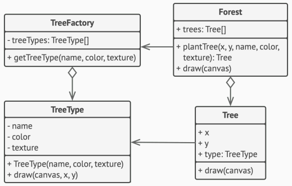
Examples:

- Layered architecture often use a facade as top-level access module
- Access parser, type checker, code generator through a compiler object
- A database facade exposes specific operations, but not arbitrary SQL operations



3.2.2. Flyweight Pattern

- Use sharing to optimize RAM usage by large numbers of fine-grained objects that share some immutable properties (intrinsic attributes)
- We cache the intrinsic data using the *factory method*, the cached objects are called flyweights

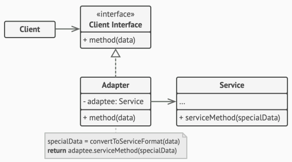



```
class TreeType is
    field name
    field color
    field texture
    constructor TreeType(name, color, texture) { ... }
    method draw(canvas, x, y) is
        // 1. Create a bitmap of a given type, color &
texture.
        // 2. Draw the bitmap on the canvas at X and Y
coords.

// Flyweight factory decides whether to re-use existing
// flyweight or to create a new object.
class TreeFactory is
    static field treeTypes: collection of tree types
    static method getTreeType(name, color, texture) is
        type = treeTypes.find(name, color, texture)
        if (type == null)
            type = new TreeType(name, color, texture)
            treeTypes.add(type)
        return type

// The contextual object contains the extrinsic part of
the tree
// state. An application can create billions of these since
they
// are pretty small: just two integer coordinates and one
// reference field.
class Tree is
    field x,y
    field type: TreeType
    constructor Tree(x, y, type) { ... }
    method draw(canvas) is
        type.draw(canvas, this.x, this.y)

// The Tree and the Forest classes are the flyweight's
clients.
// You can merge them if you don't plan to develop the Tree
// class any further.
class Forest is
    field trees: collection of Trees

    method plantTree(x, y, name, color, texture) is
      type = TreeFactory.getTreeType(name, color, texture)
        tree = new Tree(x, y, type)
        trees.add(tree)

    method draw(canvas) is
        foreach (tree in trees) do
            tree.draw(canvas)
```

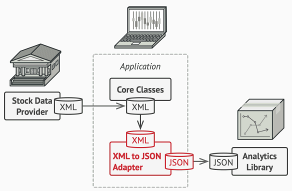
3.2.3. Adapter Pattern

- Convert the interface of a class into another interface clients expect.
- Adapter allows objects with incompatible interfaces to collaborate.

Scheme



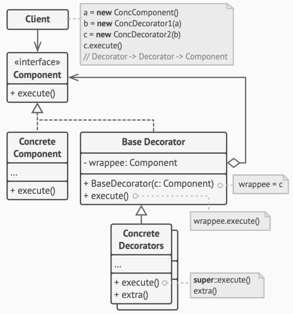
Example



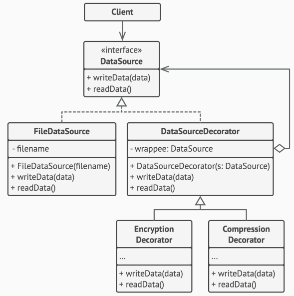
3.2.4. Decorator Pattern

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Scheme



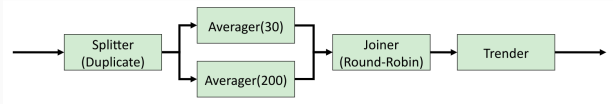
Example



3.3. Architectural Pattern

3.3.1. Pipes and Filters

- Data flow is the only form of communication between components, there are no shared state.
- Data from a source flows in a linear path through **Components** (filters)[1] that operate on the data, and **Connectors** (pipes)[2] that are connections between filters.



Properties:

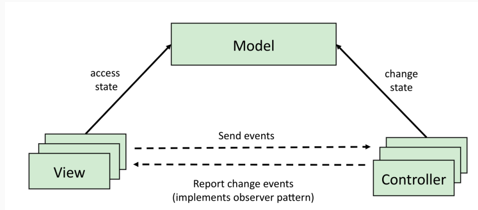
- Data is processed incrementally as it arrives, output usually begins before all input is consumed
- Filters must be independent of each other (no shared state), and don't know upstream or downstream filters.

Examples:

- Unix pipes: `grep search-text file | sort`

3.3.2. Model-View-Controller Architecture

- popular for user interfaces, it contains following components:
  a). **Model** contains the core functionality and data
  b). One or more **views** display information to the user
  c). One or more **controllers** handle user input



- Communication:
  1. Change-propagation mechanism via events ensures consistency between user interface and model
  2. If the user changes the model through the controller of one view, the other views will be updated automatically
  3. Model and view decoupled through controller

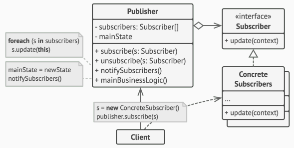
[1]Components - Read data from input ports, compute, write data to output ports; all green boxes in the figure are filters
[2]Connectors - Streams (typically asynchronous FIFO buffers), split-join connectors

3.4. Behavioral Pattern (Object Communication)

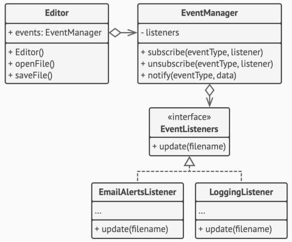
3.4.1. Observer Pattern

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Reduce coupling of generator and observer of events
  - without it, the generator and its observers would be tightly coupled, as the generator would need to maintain a direct reference to each observer and would need to know the specific interface or method to call on each observer to notify it of events.

Scheme



Example




```
// Trivial




////////////////////////////////////////////////////
class Game {
  std::vector<Player*> players;
  unsigned onTurn = 0;
  Reporter* reporter;

  public:
  Game(unsigned numberPlayers){
    for (unsigned i = 0; i != numberPlayers; ++i)
      players.push_back(new Player(i));
    reporter = new Reporter();}

  void turn(){
    reporter->turn();
    for (auto p: players) p->play(onTurn);
    onTurn = (onTurn+1) % players.size();}
}; // end of class Game

Game* constructGame(unsigned numberPlayers){
  auto game = new Game(numberPlayers);
  return game;
}

class Player {
  unsigned number;
  public:
  Player(unsigned n): number(n) {}

  void play(unsigned n){
    if (n == number){
    std::cout << number <<": I can play" << std::endl;
    } else {
     std::cout << number << ": an opponent plays, I wait."
      << std::endl;}
  }
}; // end of class Player

class Reporter {
  int turns = 0;
  public:
  void turn(){
    std::cout << "turn " << turns++ << std::endl;
  }
}; // end of class Reporter
```

```
// Observer Pattern
class Subject {
  // 1. One subject, which has a generic list of observers
  std::vector<Observer*> observers;
  public:
  // 2. 0-n observers who can subscribe to the subject.
  void subscribe(Observer* observer){
    observers.push_back(observer);
  }
 // 3. Subject notifies all observers of generated events.
  void update(){ for (auto o: observers) o->update();}
};

class Observer { public: virtual void update()=0; };

////////////////////////////////////////////////////
class Game: public Subject {
  unsigned players;
  unsigned onTurn = 0;

  public:
  Game(unsigned numberPlayers){ players = numberPlayers;}
  unsigned current_turn(){ return onTurn;}
  void turn(){ update(); onTurn = (onTurn+1) % players; }
}; // end of class Game

Game* constructGame(unsigned players){
  Game* game = new Game(players);
  game->subscribe(new Reporter());
  for (unsigned i=0; i<players; ++i)
    game->subscribe(new Player(game,i));
  return game;
}

class Player: public Observer{
  unsigned number;
  Game* game;
  public:
  Player(Game* g, unsigned num): game(g), number(num) {}

  // 4. Observer can then decide how to handle the events
  void update(){
    unsigned n = game->current_turn();
    if (n == number){
      std::cout << number <<": I can play" << std::endl;
    } else {
     std::cout << number << ": an opponent plays, I wait."
      << std::endl;}
  }
}; // end of class Player

class Reporter: public Observer {
  int turns = 0;
  public:
  void update(){
    std::cout << "turn "<<turns++<< std::endl;
  }
}; // end of class Reporter
```

3.4.2. Visitor Pattern

- Represent an operation to be performed on the elements of an object.
- Lets you define a new operation without changing the classes of the elements on which it operates

Examples: Double invocation

- where `eval()` was used within each *subclass*, replaced by a `void accept(Visitor)` instead
- a new class `Visitor` which contains a function `virtual visitSubclass(Subclass obj)`
  - instead of `eval()`, we let an Evaluator class extends the visitor, and within there, we provide concrete implementation how the visit should be done.

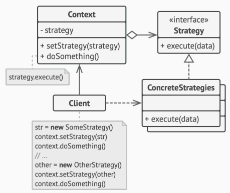
```
class Exp{
virtual double eval() const = 0;
};

class Literal: public Exp{
  double eval() const{
    return val;
}

struct Addition: public Exp{
  double eval() const
  {return left->eval() +
            right->eval();}
}

// PRE-VISITOR PATTERN
class Expression{
  ...
  double eval(){
    if (op == '=')
      return val;
    ...
  }
}
```

```
class Exp{
virtual void accept(Visitor) = 0;
};

class Literal: public Exp{
  double accept(Visitor v){
    v.visitLiteral)(this);
}

struct Addition: public Exp{
  double accept(Visitor v)
    {v.visitAddition(this);}
}

// VISITOR PATTERN
class Visitor{
  virtual void visitLiteral(Literal e) = 0;
  virtual void visitAddition(Addition e) = 0;
}

// 1.concrete implementation of how to visit
class Evaluator extends Visitor{
  double value;
  void visitLiteral(Literal e){
    value = e.value;
  }
  void visitAddition (Addition e){
    Evaluator l; e.left.accept(l);
    Evaluator r; e.right.accept(r);
    value = l.value + r.value;
  }
}

// 2.concrete for how to print
class Printer extends Visitor{...}
```

3.4.3. Strategy Pattern

- Defines a family of algorithms, puts each of them into a separate class, and makes their objects interchangeable.
- Lets the algorithm vary independently from clients that use it.
- *Based on composition: you can alter parts of the object's behavior by supplying it with different strategies that correspond to that behavior.*
- *Strategy works on the object level, letting you switch behaviors at runtime.*

Scheme

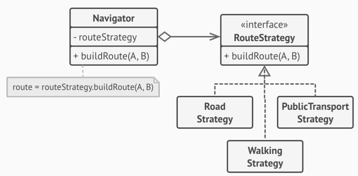


Properties

- Context (the original class)
  - ***member variable:*** must have a field for storing a reference to one of the strategies. The strategy object performs the execution `strategy.execute()`, not the context object
  - ***member function:*** The context isn't responsible for selecting an appropriate strategy. Instead, the client executes `context.setStrategy(str)`

This way the context becomes independent of concrete strategies, so you can add new algorithms or modify existing ones without changing the code of the context or other strategies.

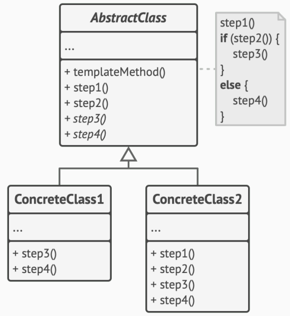
Example



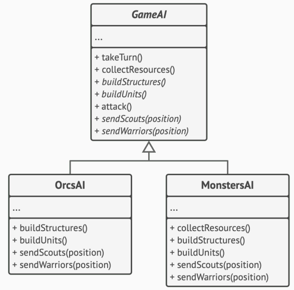
3.4.4. Template Method

- Turns a monolithic algorithm into a series of individual steps (that make up the skeleton of an algorithm) but lets subclasses override specific steps of the algorithm without changing the structure defined in the superclass.
- *Based on inheritance: it lets you alter parts of an algorithm by extending those parts in subclasses.*
- *Template Method works at the class level, so it's static.*

Scheme



Example



4. Testing

Testing

- Testing is the process of executing a program to find deviations in the program's behavior from the expected behavior as specified in requirements (functional/nonfunctional)
- A successful test should be able to:
  - 1). Have a high probability of finding an error (cannot show the absence of bugs though but only show the presence of bugs)
  - 2). demonstrate that the software appears to be working according to the specification
  - 3). collect data during testing to indicate software reliability & quality; reveal nonfunctional regressions like performance loss

4.1. Example: Google Tests

```
TEST(CompleteBranchCov, FirstIsZero) {
  double actual = average(std::vector<int> { 0, 2, 4, 10 });
  double expected = 4.0;
  EXPECT_DOUBLE_EQ(actual, expected);
}

// fetch exceptions
TEST(CompleteBranchCov, EmptyArray) {
  EXPECT_THROW({
      average(std::vector<int> {});
  }, std::logic_error);
}
```

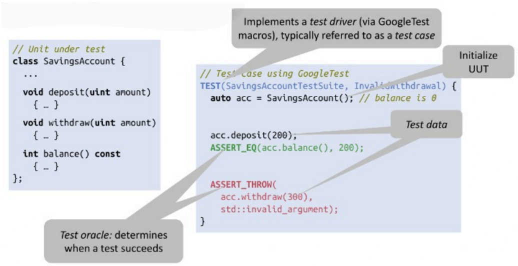
4.2. Test Stages

Test Stages	Goal & Content

Unit Test

To confirm that the subsystem is correctly coded and implements the required functionality.

- testing individual subsystems, including functions, (group of) classes



- Use parameterized unit tests to achieve a reasonable test coverage.
  - 1). Parameterize with input data and expected output data (provide expected output data with `INSTANTIATE_TEST_SUITE_P`)
  - 2). Validate obtained output by asserting relevant properties (`std::is_sorted()`, `ASSERT_EQ(...,...)`, `ASSERT_LE(...,...)`, `ASSERT_FALSE(j == data.size())` etc.)
  - 3). Validate against a master solution (eg. `falcon_sort(data)` against `std::sort(data)`)

Example:

```
// 1. Test fixture providing initialized account
class ParametricSavingsAccountTests: public testing::TestWithParam<std::tuple<int,
int>>{
  protected:
    SavingsAccount acc;
}

// 2. Test data
const std::vector<std::tuple<int,int>> DepositThenWithdraw_data = {
  {99,1},
  {50,50}
};

// 3. Parametric test driver
TEST_P(ParametricSavingsAccountTests, DepositThenWithdraw){
  auto [deposit_amount, withdraw_amount] = GetParam();

  acc.deposit(deposit_amount);
  acc.withdraw(withdraw_amount);

  ASSERT_EQ(acc.balance(), deposit_amount - withdraw_amount);
}

// 4. Instantiate test driver for each test data entry
// data represented as a vector of pairs of {deposit, withdraw} amount
INSTANTIATE_TEST_SUITE_P(
DepositThenWithdraw,
ParametricSavingsAccountTests,
testing::ValuesIn(DepositThenWithdraw_data)
);
```

- Test execution (for other test stages similarly)
  1. Regression testing: re-running tests to ensure the software still performs as expected after a change.
  2. Automate as much as possible

Integration Test

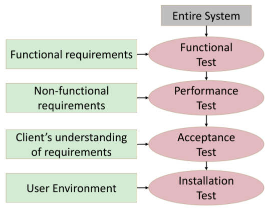
To test interfaces between subsystems;

- testing groups of subsystems and eventually the entire system
- different strategies to decide the call hierarchy (eg. big bang that includes all components for testing or bottom-up/top-down integration)

System Test

To determine if the system meets the functional and nonfunctional requirements;

- testing the entire system



Acceptance Test

To demonstrate that the system meets customer requirements and is ready to use;

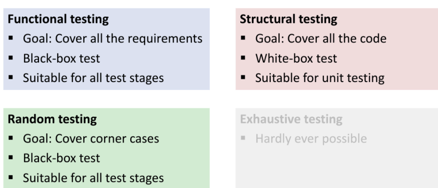
- performed by the client
- alpha test: The client uses the software at the developer's site; software used in a controlled setting, with the developer ready to fix bugs
- beta test: conducted at client's site; software gets a realistic workout in the target environment

Independent Test

To efficiently find flaws and failures;

- Testing done by independent test engineers to prevent author bias, though while coding developers should also write down unit & integration tests
- testers and developers collaborate in developing the test suite
- the testing team is not solely responsible for the software quality, the quality should be assured by a good software development process

4.3. Overview: Testing Strategies



Test Strategies	Content

Functional Testing

Test each case of the specification, eg. testing for all cases of discriminant in $ax^2 + bx + c = 0$;

- **black-box testing:** tests a unit against its requirements, note that tests are not derived from code or design
- **Advantage:** Tests can be written without access to the code, or having to understand it; Tests are suitable for all possible implementations; Good chance of revealing incorrect or missing functionality
- **Disadvantage:** Often not effective for detecting coding errors, e.g. buffer overflows, memory management, faulty optimizations and design flaws, e.g. too much coupling; Quality of the resulting test suite difficult to access automatically;

Structural Testing

To create tests that exercise as much of the code as possible

- **white-box testing:** look at code or design to derive tests from UUT; Use design knowledge about system structure, algorithms, and data structures to determine test cases
- **Advantage:** Enables code coverage metrics and automated computation; Effective at detecting coding errors (since the code needs to be studied to derive test cases); Exploring all execution paths may reveal arbitrary coding errors; Closer look at code may reveal additional problems (design,

performance, security, ...)

- **Disadvantage:** Requires knowledge about the internals, which testers and clients shouldn't need to have → not well-suited for system tests; Time-consuming to increase or achieve total coverage; Implementation-specific tests more likely to become obsolete over time; Less effective at uncovering only partially implemented requirements

4.4. Functional Testing

Equivalence Classes

- An equivalence class is a collection of test cases/inputs that provoke a similar functional behavior in the unit under test.

1. Partition the total value space
2. Boundary testing: select elements at the edge cases of each equivalence class
3. Combine concrete inputs for testing for each equivalence class

- Performing functional testing is dividing input values using case distinctions

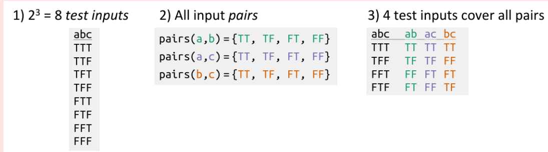
	Two solutions	One solution	No solution
Linear equation		a = 0 and b ≠ 0	a = 0, b = 0, and c ≠ 0
(Truly) quadratic equation	a ≠ 0 and b^2-4ac > 0	a ≠ 0 and b^2-4ac = 0	a ≠ 0 and b^2-4ac < 0
Invalid input	a = 0, b = 0, c = 0		

Semantic Constraints

- Use problem domain knowledge to remove unnecessary combinations by enforcing semantic constraints on combinations
- Advantages:
  - potentially reduces no. test cases
  - increases coverage by identifying semantic equivalence classes (not just by looking at involved types)
  - may uncover issues with specifications
- Disadvantages:
  - still, too many combinations remain

Pair-wise Combinations (combinatorial testing for two or less inputs)

- **Motivation:** Empirical evidence suggests that most bugs do not depend on the interaction of many variables. Most errors are triggered by interactions of 2-3 variables.
- **Goal:** To focus on all possible combinations of each pair of inputs
- **Examples:** Given `voi fun(bool, a, b, c)`



- **Advantages:**
  - *Complexity:* No. test cases grows logarithmically in n and quadratic in d - $O(\log(n), d^2)$, with n as no. parameters and d as test values per parameter. d can be influenced by the tester
  - This reduces the number of tests necessary to detect bugs in the code reliably; Suitable when many system configurations (hardware, OS, database, application server, etc.) need to be tested.

4.5. Structural Testing

Structural Testing

1. Control flow testing (control-flow graphs)
2. Coverage (statement, branch, path, loop)

- **Approach:** white-box testing
- **Goal:** cover a large portion of the unit under test's code

4.5.1. Control-flow graphs (CFGs)

Control-flow graphs (CFGs)

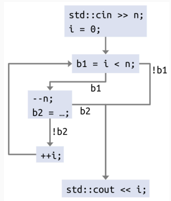
- Control-flow graphs (CFGs) are typical internal representations in code analysis tools, including compilers where:

1. Nodes are basic blocks
2. Edges between basic blocks bb_1, bb_2 with condition c denote that:
   - the execution after the last statement of block bb_1 continues with the first statement of block bb_2 if condition c holds.
3. A node without an incoming edge is an entry node; without an outgoing edge is an exit node and a node can be made unique by introducing dedicated, empty blocks

- The CFG can serve as a quality criterion for test cases: the more parts (nodes, edges, paths) are executed, the higher the chance of uncovering a bug.

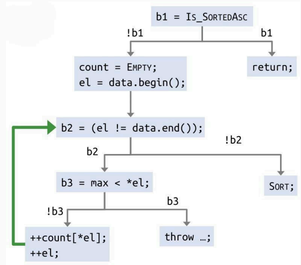
Example 1:

```
std::cin >> n;
for (i = 0; i < n; ++i){
  --n;
  if (...) break;
}
std::cout << i;
```



Example 2:

```
// IS_SORTEDASC
if (
  is_sorted(data.cbegin(), data.cend())
) return;

// count = EMPTY
auto count
  = std::vector<unsigned>(max+1,0);

for (auto elem: data){
  if (max < elem)
    throw invalid_argument("...");
  ++count[elem];
}

// SORT
auto it = data.end() - 1;
for (unsigned i = 0; i < count.size(); ++i){
  for (unsigned c = count[i]; 0 < c; --c){
    *it = i;
    --it;
  }
}
```



4.6. Coverage

Coverage

- Coverage is a good way of measuring the adequacy of tests
- white-box approach, computed relative to control-flow graph
- statement and branch coverage are standard, other measures exist
- ⚠ High coverage does not imply well-tested code (bugs could still exist), but low coverage implies the code is not well-tested

  DON'T BLINDLY OPTIMIZE FOR COVERAGE NUMBERS
- Perfect coverage (exhaustive testing) is infeasible due to loops, large state space etc.

)

Statement

- Assess the quality of a test suite by measuring how many statements of the CFG it executes (one can detect a bug in a statement only by executing the statement)

$$\text{Statement Coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}}$$

- can also be defined on basic blocks
- **Example:** achieve 100% statement coverage with three test cases.

1. `d = {1,3,2}` (80% statement coverage as this single test case executes 8 out of 10 statements), `m=3`;
2. `d = {1,3,2}`,`m=0`;
3. `d={1,2,3}`, `m=3` (this uncovers the bugs (??))

Branch

- Test all possible branches (edges with conditions) in the control flow; Most widely-used adequacy criterion in industry

$$\text{Branch Coverage} = \frac{\text{Number of executed branches}}{\text{Total number of branches}}$$

- **Advantage:** leads to more thorough testing than statement coverage
  - complete branch coverage implies complete statement coverage
  - But "at least n% branch coverage" does not generally imply "at least n% statement coverage"

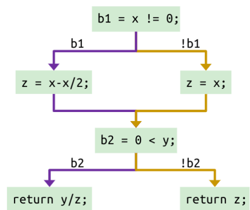
Recall example 4 with the invalid file problem: We still have the same CFG, and we can use two test cases to cover all branches

1. `filename` denoting a readable file
2. `filename` denoting a non-existing file

=> The second test case exposes the bug

- **Limitation:** Possible to have 100% branch coverage but still cannot expose the bug

```
// 100% branch coverage with two tests
// 1. x=1,y=1; 2. x=0,y=0 => but if x=0,y=1 then z=0 while y/z executes
int foo(int x, int y){
  int z;
  if (x!=0) z = x - x/2;
  else z = x;
  if (0<y) return y/z;
  else return z;
}
```

- **Solution:** Cover all statements under all conditions, cover all possible branch combinations.
  - Above we covered all branches but not through all branch combinations. Here adding tests 3. `x=1,y=0`; 4. `x=0,y=1` *[This will hit the bug]*. In all we cover the paths b1→!b2 and !b1→b2
  - 

Path

- **Goal:** find test cases that cover every possible unique path through the CFG.

$$\text{Path Coverage} = \frac{\text{Number of executed paths}}{\text{Total number of paths}}$$

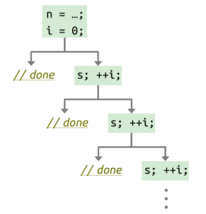
- A path is a valid sequence of CFG nodes $n_1,\ldots,n_k$ (basic blocks) that start with the entry node and end with the exit node. There are edges between n_i and n_{i+1}
- *Path coverage is more thorough than statement & branch, but finding test cases can be even more time-consuming*

⚠ Path coverage cannot be measured when the code has an unknown no. loop iterations. Complete path coverage is generally not feasible for loops, loop coverage is an alternative.

- **Limitation:** Loops introduce arbitrarily many CFG paths, here branch conditions are omitted for brevity

```
n = …; // user input
for (i = 0; i < n; ++i)
  s;
```

```
n = …;
i = 0;
if (i < n) { s; ++i; }
if (i < n) { s; ++i; }
if (i < n) { s; ++i; }
…
if (i < n) { s; ++i; }
…
```



Loop

$$\text{Loop Coverage} = \frac{\text{Number of executed loops with 0, 1, and more than 1 iterations}}{\text{Total number of loops * 3}}$$

For 100% loop coverage, each loop must be executed

1. exactly zero times
2. exactly one time
3. more than once consecutively

- Loop coverage should be combined with other adequacy criteria such as statement or branch coverage

5. Formal Methods

Symbolic Execution

- compute the symbolic constraints per path, and then solve these constraints such that we have concrete inputs that explore all paths
- **Path exploration strategies:** to avoid exploring infeasible paths and wasting our time, symbolic execution engines may include heuristics that
  1. Solve constraints at every branch point to quickly obtain the first results
  2. Apply different exploration strategies (eg. BFS, DFS, prefer shallow paths, complex conditions...)

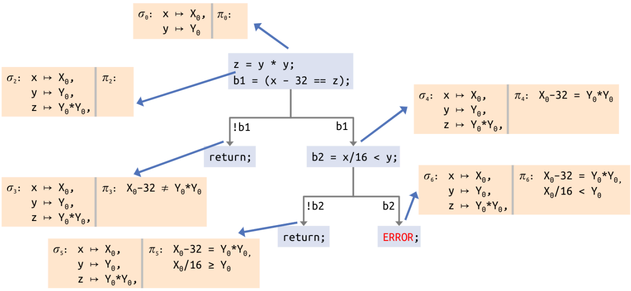
Concolic Execution = Concrete (Testing) + Symbolic

- assign concrete values to symbolic inputs and execute the given program both concretely and symbolically at the same time
- **Concretization:** In case the SMT solver fails to solve any constraint, constraints of a path can be simplified by plugging user-provided initial values into one of the symbols.

5.1. Example: Symbolic Execution

1. A symbolic state σ
   - maps variables to symbolic expressions
   - is used to evaluate program expressions to symbolic expressions
   - is updated by assignment statements
2. Path conditions π are the conditions under which a path is taken
   - we have a symbolic state per program point

We can solve the final constraint sets at the bottom-most leaves



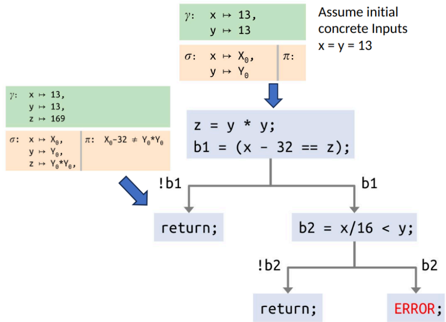
Alloy = Logic + Language + Analysis

- Alloy is a formal modeling language based on set theory
- An alloy model specifies a collection of constraints of a model and finds structure that satisfy them
  - generate sample structures
  - generate counterexamples for invalid properties
  - visualize structures

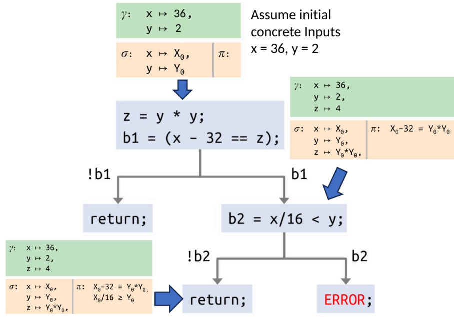
5.2. Concolic Execution

- (a). The !b1 route will be taken given the initial inputs `x = y = 13`.
- (b). Afterwards, the condition is negated: `X0-32 = Y0*Y0`. The solver is queried for a model and returns `X0 = 36, Y0 = 2`. It reaches the b1 and then the !b2 branch.
- (c). Negating !b2 and querying the solver yields new inputs, e.g. `X0 = 81 and Y0 = 7`. The concolic execution will take the b1 and then the b2 branch.
- All paths have been explored.

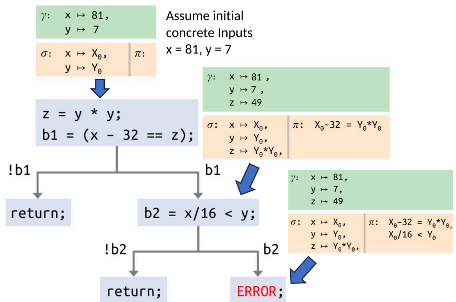
(a).



(b).



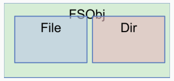
(c).



6. Alloy Cheatsheet

6.1. Signatures

General	`// x ⊂ A × B` `sig A { x : B }`

Extend 	• If `A` and `B` each extends `C`, then `A` and `B` are disjoint
	`sig name extends superclass {...}`
	`// Example: Subtyping`
	`sig FSObject {}`
	`sig File extends FSObject {}`
	`sig Dir extends FSObject {}`

Abstract 	`abstract sig name {...}`
	`abstract sig FSObject { parent : lone Dir} // parent ⊂ FSObject × Dir`
	`sig File extends FSObject {}`
	`sig Dir extends FSObject { contents: set FSObject} // contents ⊂ FSObject × FSObject`
	`one sig Root extends Dir`

Subset	N.B. Subset signatures are not necessarily pairwise disjoint, and may have multiple parents.
	`sig name in sup {...}`
	`sig name in sup1 + sup2 + ... {...}`

6.2. Paragraphs

Facts	``// `Name` is optional``
	`fact name { formulas}`
	`// Example`
	`fact {all n: Node | n!=n.next} // ∀n | (n,n) ∉ next`

Predicates	Predicates are either true or false; they are named, parameterized formulas.
	`pred pred_name { F}`
	`pred pred_name [x1: e1, ..., xn: en] { F}`
	`// 1. Use predicate in a function (see below)`
	``// 2. Find instance of a predicate uses `run pred_name`, function can also be run but it is less common``

Functions	The body expression E is evaluated to produce the function value; the bounding expression e describes the set from which the result is drawn.
	`fun name [ x1:e1, ...] : e { E}`
	`// Example: returns the set of all non-occupied seats in flight f`
	`fun freeSeats [f: Flight]: set Seat{`
	`{s : f.seats | no f.seating[s]}`
	`}`
	`// Example: FSObject using predicate`
	`fun leaves[f: FSObject]: set FSObject{`
	`{x: f.*contents | isLeaf[x]}`
	`}`

Assert	Unlike predicates, assertions don't bind arguments `assert name { F }`
Check Assertions	Use check to look for counter-examples `check name for 2 but 1 sig1, 5 sig2`
Run	Use `run` to request an instance satisfying the predicate; One can also specify scope. Defaults to 3. `run name for 2 but 1 sig1, 5, sig2`
Let	`let decl, decl2 ... | expression` `let decl, decl2 ... { formulas }`

6.3. Declarations

Fields of signatures, function arguments, predicate arguments, comprehension variables, and quantified variables all use the same declaration syntax:

Multiplicity of fields	• `no`: empty set (not some e); $e = \emptyset$ • `some`: non-empty set (not no e); $e \neq \emptyset$ • `one`: singleton set (default); $\|e\| = 1$ • `lone`: singleton or empty set; $\|e\| \leq 1$

	• set: zero or more elements (any set);

Quantification

- Constraint sets/relations

```
all x : e | F // for all x in set e, fact F holds
all x : e1, y: e2 | F
all x, y : e | F // for all x,y (duplicate possible) in e, fact F holds
all disj x, y : e | F
```

- Specifying element amounts

```
all x : e | F
some x : e | F
no x : e | F
lone x : e | F
one x : e | F
```

- Acyclic
  - Contents-relation is acyclic `no d: Dir | d in d.^contents`

Special Expressions

- Empty set: `none`
- Universal set: `univ`
- Identity function: `iden`

Distinct Values

Using disjoint `disj`, requires distinct `S` atoms to have distinct `f` values

- `sig S { f : disj e}`
- `all a, b : S | a != b implies no a.f & b.f`
- `all disj a, b : S | disj [a.f, b.f]`

Relations

- `r : e1 -> e2`

Bounding expression may denote a relation

```
r : e1 -> one e2 // Total function (`-> one`)
r : e1 -> lone e2 // Partial function (`-> lone`)
r : e1 one -> one e2 // Bijection (`one -> one`)
```

Ternary Relation

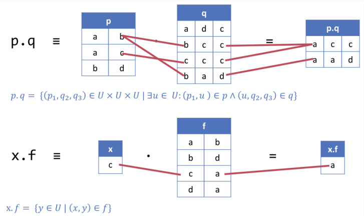
enrollment		
U0	S0	P1
U0	S1	
U1	S1	P2
U1	S2	P3
U2	S3	P1

```
// map ⊂ A × I × B
sig A{ map: I -> B} // "->" is Cartesian product
a.map[i]            // assuming a, i are singletons {b ∈ B | (a,i,b) ∈ map}


// Example: enrollment ⊂ University × Student × Program
sig University{
  students: set Student,
  enrollment: students set -> one Program
}
```

6.4. Element Retrieval & Operators

Joins

- relational join: `a.x` meaning {b ∈ B | (a,b) ∈ X}



- box join: `x[a] := a.(x)`

Dot binds tighter than box, so `a.b[c] ≡ (a.b)[c]`

NOTE: Left associativity, equivalent statements

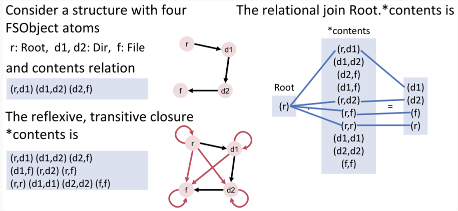
- `a.b.c`
- `(a.b).c`

Unary

- transpose: `~r`
  - `~R = { (r2 , r1) ∈ U × U | (r1, r2) ∈ R}`
  - eg. In an undirected graph, for each edge, the reverse edge is also contained in the graph `g.adj = ~(g.adj)`
- positive transitive closure: `^r := r + r.r+ r.r.r + ...`
  - eg. descendants `= {a,d: Person | d in a.^children}`
  - eg. predicate describes a graph free of cycles, using `n.*(g.adj)` would be wrong!

```
pred cycleFree(g: Graph){
  no n: g.nodes | n in n.^(g.adj)
}
```

- reflexive transitive closure: `*r := iden + ^r`

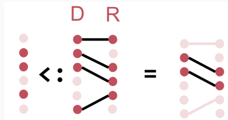


Set operations

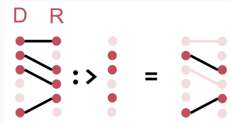
- Union: `a + b`
- Intersection: `a & b`
- Difference: `a - b`

Restrictions

- domain restriction: `X <: R` $\{(r1, r2, \ldots, rn) \in R \mid r1 \in X\}$

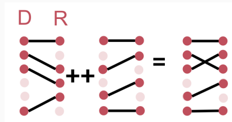


- range restriction: `R :> X` $\{(r1, r2, \ldots, rn) \in R \mid rn \in X\}$



Override

- relational override: `p ++ q P - (domain[q] <: p) + q`
- Set representation: $\{p \in P \mid p_1 \neq q_1\ \forall q \in Q\} + q$



Cardinality

- Number of tuples in a: `#a`
- Equals: `=`

```
// eg. sum of integer expression ie for all singletons x drawn from e
sum x: e | ie

// eg.2 all bags have 3 or less marbles
all b: Bag | #b.marbles =< 3

// the sum of the marbles across all bags equals the total no. marbles
#Marble  = sum b: Bag | #b.marbles
```

Logics

- Negation: `not`, `!`
- Conjunction: `and`, `&&`
- Disjunction: `or`, `||`
- Implication: `implies`, `else`, `=>`
  - `boolean implies expression`
  - `boolean implies expr1 else expr2`
- Bi-implication: `iff`, `<=>`

NOTE:

1. `p=>q=>r` equivalent to `p=>(q=>r)`
2. `else` binds to the nearest possible `implies`, equivalent statements:
   - `p=>q=>r else s`
   - `p=>(q=>r else s)`

Arithmetic

Apply only to integer expressions

- `plus`, `minus`, `mul`, `div`, `rem`

Equivalent statements: `plus[x][1]`, `plus[x,1]`; `x.plus[1]`, `1.(x.plus)`

6.5. Dynamic Behavior

```
// steps as fact
fact execution {
  // initial state is the first in order
  init[FileSystem]

  // states created via one of the ops
  // removeAll, add etc. are operations
  always
  { some o: FSObject | removeAll[FileSystem,o] }
  or
      {  some  o:  FSObject,  d:  Dir  |
add[FileSystem,o,d] }
  }
}
```

```
// steps as predicate
pred Steps {
  init [first]
  and all s: State - last |
      some  sem:  Semaphore,  t:  Thread  |
acquire[s,s.next,t,sem]
    or
      some  sem:  Semaphore,  t:  Thread  |
release[s,s.next,t,sem]
}

//∄No  state  s.t  all  threads  wait  for  a
semaphore.
assert StepsDoNotCreateDeadlock{
  Steps =>
  no s: State | all t: Thread | some s.waits[t]
}
```

6.5.1. Using Temporal Notion in Alloy 6

Alloy	LTL
after (or ' or ;)	X (or ○)
always	G (or □)
eventually	F (or ◊)
until	U (or U)

```
// Example: Using after,always, eventually and
until
some n: Node {
  n in b.loc.edges
  after b.loc = n
}

some b: Ball | always move[B]
all n: Node | eventually b.loc = n
move[b] until b.loc = Last
```

```
// Pre Alloy6
pred move[from: State]{
from.next.loc != from.loc
}
fact "always move"{
First.loc = Ping
all p: State – Last | move[p]
}
pred show{}
run show for exactly 5 State
```

```
// Post Alloy6
pred move[from: State]{
from.loc' != from.loc
}
fact "always move"{
State.loc = Ping
always move[State]
}
pred show{}
run show for 5..5 steps
```

6.5.2. Mutable Relations

Using `var` keyword on a field to specify mutability

```
sig E {}

sig Array {
  length: Int, // relation length is static
  var data:    // with "var", the relation `data` is
mutable
        {i: Int | 0 <= i && i < length } -> lone E
}{
  0 <= length
}
```

6.6. Equivalent Temporal Statements

```
// 1. equivalence of "eventually X" and "true until X"
assert Eventually {
  {eventually Variable.col = Red} <=>
  {Variable.col in Color until Variable.col = Red}
}
// 2. equivalence of "always X" and "not eventually not X"
assert Always {
  {always Variable.col = Red} <=>
  not {eventually not Variable.col = Red}
}
```

6.7. Traces

Define the temporal behavior of the model

1. Initialize first state using `init[state]`
2. Constraint subsequent states using LTL formulas, such as `always` or `until`

```
fact traces {
  init[state]
  always{
    (some ... | op1[s,...]) or
    ...
    (some ... | opn[s,...])
  }
}
```

7. Appendix: UML Sequence Diagrams

The Unified Modeling Language UML

UML is a modeling language
- Using text and graphical notation
- For documenting specification, analysis, design, and implementation

Draw a sequence diagram for the following use cases:

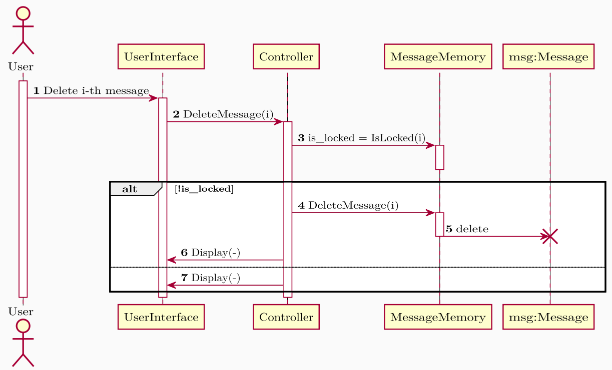
Use case 1: Delete the message.

a. User: The user asks the system to delete the i-th message.
b. System: The system checks if the message is locked (extension point).
c. System: The message is not locked, so the system deletes the message and notifies the user.

Use case 2: Fail to delete the message (extends use case 1).

a. ...
b. ...
c. System: The message is locked, so the system displays an error to the user.

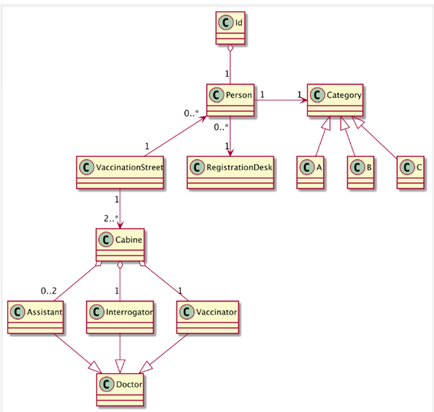
Solution:



7.1.1. Example: Vaccination

A Vaccination Street contains (→) two or more Cabines and an arbitrary number of Persons. A Doctor can have one and only one of the following roles: an Interrogator asking medical questions, a Vaccinator performing the injection, or an Assistant. A cabine has (◇-) exactly one vaccinator and interrogator respectively and up to two assistants. A Person has a unique Id, is assigned to (→) a Registration Desk and belongs to a Vaccination Category. The vaccination category is either A,B,C.

Other syntax: Each heater has access to (◆-) one temperature sensor

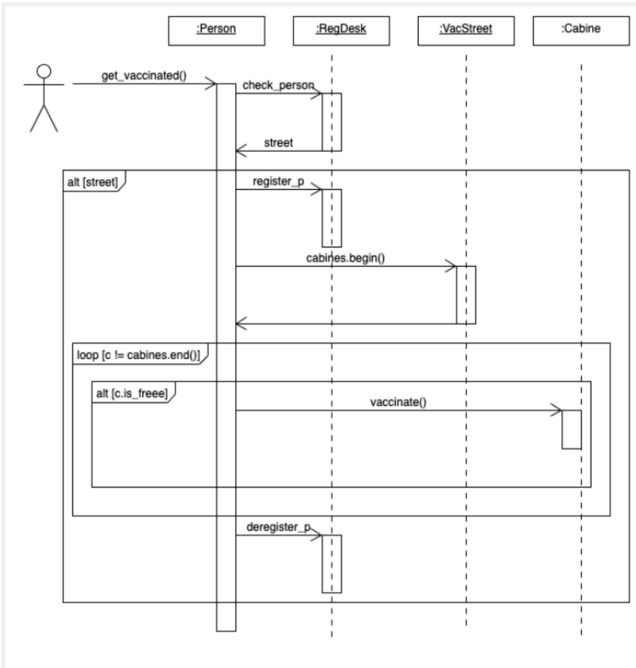


(a) Draw a sequence diagram that depicts a person getting vaccinated

```
class Person;
class Cabine {
  public:
    Cabine();
    bool is_free = true;
    void vaccinate(Person* person);
};

class VaccinationStreet {
  public:
    VaccinationStreet();
    std::vector<Cabine> cabines;
    std::vector<Person> persons;
};

class RegistrationDesk {
  public:
    void register_p(int person_id);
    void deregister_p(int person_id);
   VaccinationStreet* check_person(int person_id);
};

class Person {
  public:
    int person_id;
    bool is_vaccinated = false;
    RegistrationDesk* registration_desk;

  void get_vaccinated() {
      VaccinationStreet* street
     = registration_desk->check_person(person_id);

      if (street)
      {
         registration_desk->register_p(person_id);
          for (auto &c : street->cabines)
          {
              if (c.is_free) c.vaccinate(this);
          }
                                  registration_desk-
>deregister_p(person_id);
      }
  }
};
```

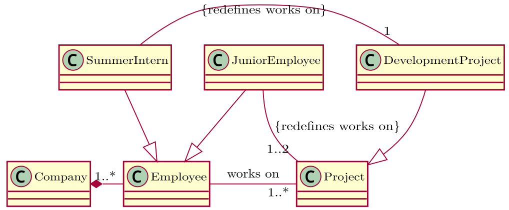
Hints:

1. Draw out the users, identify objects that are used, and list the corresponding classes in boxes. Use a pencil to full-sized demo boxes for life-scope.
   - eg. `registration_desk` is created before `street` `VaccinationStreet* street = registration_desk->check_person(person_id);`
2. Draw arrows for member functions; Arrows of member functions always end in the class of objects themselves, and start from the class where it gets called. Adjust the size of the life-scope boxes.
   - Note that for if, for, and while conditions, the arrow for directions is double-sided!
3. Identify scopes `alt` and `loop`, the scope needs to include all methods that are called within the code scope. Add conditions such as `alt [c.is_free]` to the scopes. Use dashed lines (`---`) to divide cases when needed.



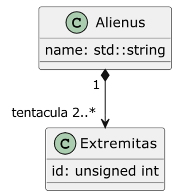
7.2. Mapping Models to Code

Company - Class Diagram




```
#include <memory>
#include <vector>

class Project { public: /* ... */ virtual ~Project() {} };
class DevelopmentProject : public Project { /* ... */ };

// generalization/superclass
class Employee {
 public:
  // ...
  virtual std::vector<std::shared_ptr<Project>> GetProjects() const;
  virtual void AssignProject(std::shared_ptr<Project> project);
};

// specialization/subclass
class SummerIntern : public Employee {
 public:
  // ...
  virtual std::vector<std::shared_ptr<Project>> GetProjects() const {
    return { project_ };
  }

  virtual void AssignProject(std::shared_ptr<Project> project) {
    if (project_)
      throw std::logic_error("Cannot assign multiple projects");

    auto development_project =
             std::dynamic_pointer_cast<DevelopmentProject>(project);
    if (!development_project)
      throw std::logic_error("Cannot assign non-development project");

    // Type system prevents us from assigning a pointer to a non-development project.
    project_ = development_project;
  }

 private:
  std::shared_ptr<DevelopmentProject> project_;
};
```

7.2.1. Past Exam




```
struct Extremitas {
  unsigned int id;
};

struct Alienus {
  std::string name;
  std::vector<Extremitas> tentacula;
};
```