UCLA CS130 Software Engineering Fall21 Review Note: Final

By Patricia Xiao

UML Diagrams

Static / Structure Modeling: fixed, code-level

- Class Diagrams
- etc. (e.g. Component Diagrams)

Dynamic / Behavioral Modeling: capturing execution of the system

- Use Case Diagrams
- Sequence Diagrams
- State Chart Diagrams
- etc. (e.g. Activity Diagrams)

Class Diagrams

Models: high-level class relations
Components:

- Class (rectangle)
  - Upper section: name of the class
  - Middle section: attributes (type, visibility)
  - Bottom section: methods (type, visibility)
- Relations (links between classes): Dependency, Association, Aggregation, Composition, Generalization, Realization

Class Diagram: Visibility Symbols

Public (+)	Private (−)	Protected (#)
Package (∼)	Derived (/)	Static (underlined)

Class Diagram: Multiplicity Definition

Multiplicity (Cardinality)
Of a **class**: A number in the upper right corner of the component; the number of objects at runtime; usually omitted and by default > 1.
Of a **relation**: Placed near the ends of an edge, indicating the number of instances of one class linked to an instance of the other class on the other side of the edge.

Class Diagram: Multiplicity Symbols

n	exactly n	$m..n$	at least m, at most n
$*$	many	$1..*$	at least one, could be more
0..1	zero or one	0..0	must be empty

Class Diagram: Relations

From weak to strong, from general to specific:

- Dependency (uses) — A uses B (dashed line pointing from A to B)
- Association (has-a) — A has a field of B object (solid line pointing from A to B)
- Aggregation (owns) — satisfies iff
  - A has a field that is a list of B objects

  (solid line pointing to B with an **unfilled** diamond at the A end / association end)
- Composition (part-of) — satisfies iff
  - A has a field that is a list of B objects
  - B object can't live outside A

  (solid line pointing to B with an **filled** diamond at the A end / association end)
- Generalization (is-a) — B extends A / sub-classing (close-headed solid line pointing to A)
- Realization — B implements A / sub-typing (close-headed dashed line pointing to A)

Use Case Diagram

Specify: Actors, System (scenario), Goals
Models: high-level **interactions**
Components:

- Actors (stick figures) – role (one user can have multiple roles)
- Use Cases (ovals) – scenario
- Relations (edges): association, inclusion, extension, generalization

Actors are **not** directly interacting with each other.

Use Case Diagram: Relations

Association

- actor – case (undirected solid line)
- case – case (dashed line with arrow)
  - inclusion (e.g. *ride* ≪ include ≫ *push button*, arrow pointing to *push button*)
  - extension – exceptional variation (e.g. *derail* is an ≪ exception ≫ of *ride*, arrow pointing to *ride*)

Generalization/Specialization (close-headed arrow pointing to more general one); e.g. *Synchronize Data* generalize *Synchronize Data Wirelessly*

Sequence Diagram

Models: **communication** between elements
Belongs to **Interaction** Diagrams (include: Sequence diagrams, Communication diagrams, Interaction overview diagrams, Timing diagrams)
Components:

- Class Roles / Participants (top-row) / Actors
  - instance_name : Class_Type
  - not necessarily an object in the system, e.g. can be human actors.
- Activation or Execution Occurrence (dispatch: solid black dot, destroy ≪ destroy ≫)
- Messages (horizontal arrows)
  - Method Invocation (solid line with arrow)
  - e.g. a:A point to b:B with text execute(0), then it means a (of class A) calls b.execute(0), b is of class B.
  - Return value via dashed line pointing back
- Lifelines (dashed vertical lines)
  - Invocation Lifetime: vertical rectangles
  - can be nested across actors, and threads within a single actor
- **Loop** (while / for, [condition]) / **Alt** (if-then-else, [if-condition] – horizontal dashed line – [else]) / **Opt** (if-then, [if-condition]) / Par / Region; All shown as wrapped in a rectangle.

Seq Diagram: Invocation Lifetime v.s. Lifetime

When a:A create an instance of b:B at run time, we draw the rectangle with text content b:B **at the height** where a:A invokes it.
Then it starts to live. When a:A create an instance of B named b, we depict it by letting a:A pointing to a newly-created b:B column via dashed line and text: create(params); where params are the parameters needed for instantiate an object of class B.
Invocation Lifetime is not **Lifetime**.
Lifetime is represented by the dashed line, invocation lifetime is represented by the thin vertical rectangle along the dashed line.

Seq Diagram: Class Name and Type

If name of an object of class A is unknown, it is okay to leave it blank, e.g. : A.

State Chart Diagram

Models: high-level **state behaviors** of objects
Components:

- Initial State (black filled circle) – start
- Transition (solid arrow)
  - *trigger [guard] / effect*
  - *trigger* if *guard*, make *effect*
  - e.g. Somewhere is a Door's State Machine: *use key [door locked] / [door → unlock]*
- State (rounded rectangle) – of object
- Fork (rounded solid rectangular bar) – 1 incoming arrow, n outgoing arrows; represent splitting into concurrent states.
- Join (rounded solid rectangular bar) – n incoming arrows from the joining states, m outgoing arrow towards the common goal states; multiple states concurrently converge into one on the occurrence of an event or events.
- Self transition (solid line w. arrow pointing back to itself) – the state of the object does not change upon the occurrence of an event
- Composite State (rounded rectangle) – wrapping around a lot of other states
- Final state (black filled circle within a circle) – the final state in a state machine diagram

UML Diagram: Translations

format	Class	UC	Seq	State	Code
Class	N/A	✗	✗	✗	✓
UC	✗	N/A	✗	✗	✗
Seq	✗	✗	N/A	✗	✓
State	✗	●	✗	N/A	●
Code	✓	●	✓	✓	N/A

UC represents Use Case Diagram, Seq represents Sequence Diagram. Code refers to Java-style pseudo code. The meaning of the marks are listed below:

✓	sufficient (for the row) to transform to (the column)
●	transformation (from row to column) is doable but needs some extra clarification
✗	very unlikely to directly transform (the row) to (the column)

Software Design Principles

- **Information Hiding** (IH)
- Low Coupling (LC): Reduce the dependencies between modules (classes, packages, etc)
- High Cohesion (HC): A module contain functions that logically belong together.
- Separation of Concerns (SoC): a single concern is easily separated from the rest of concerns.
- etc. (e.g. Law of Demeter (LoD), Abstraction, Liskov Substitution Principle, ...)

There are many different principles. In this class we focus on information hiding.

Modularization

- Decomposition of a software system into multiple independent modules.
- Easy to interpret & maintain & code-reuse, etc.

Parnas' Information Hiding (IH) Principle

- A principle for breaking program into **modules**.
- API should (1) only contain design decisions unlikely to change (2) do not reveal any volatile information.
- Makes anticipated changes affect modules in an isolated and independent way.

Information Hiding (IH) Principle: Conclusion

Information hiding principle is:

- an analysis of how changes will affect existing code
- and assessment of changeability.

Modularization: Practice

Identify the Modules': **name**, **role**, **input**, **output**.
Changeability Assessment: for **different scenarios**, which module / which module's API(s) need to be changed.
Code Critique:

1. What information is hidden (by XXX Module)?
2. Changes you anticipate? (any new features you may want for the system)
3. Readability and comprehensibility? (e.g. consistent arguments, self-explanatory coding, etc.)
4. Capability to support independent work assignment? (low coupling)

Modularization: Different Ways to Achieve

Functional decomposition (Flowchart approach)

- Each module corresponds to each step in a flow chart.

Information Hiding (IH)

- Each module corresponds to a design decision that are likely to change and that must be hidden from other modules.
- Interfaces definitions were chosen to reveal as little as possible.

Design Patterns: Categories

Creational Design Pattern

- **Factory Method**: defines an interface for creating an object but lets subclasses decide which class to instantiate; lets a class defer instantiation to subclasses.
- **Abstract Factory**: provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Singleton**: ensures a single object creation, and it must be globally accessible.
- etc. (e.g. Prototype)

Structural Design Pattern

- **Adaptor**: adapts legacy code to a target interface.
- **Façade**: simplifies complex interfaces of multiple subsystems.
- **Flyweight**: share common resources by separating usage contexts from used objects.
- etc. (e.g. Composite)

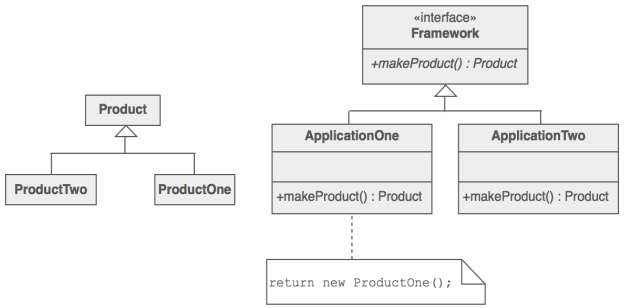
Behavioral Design Pattern

- **Strategy**: defines a family of algorithms, encapsulates each one, and makes them interchangeable at runtime; lets the algorithm vary independently from clients that use it.
- **Observer**: defines one-to-many dependency between objects, when the subject changes state, all of its observers are notified and updated.
- **Mediator**: defines an object that encapsulates how a set of objects interact, encapsulates many to many dependencies between objects, centralizing control logic, reduces the variety of messages.
- **Command**: decouples a receiver object's actions from invokers.
- **Template Method**: set a common workflow where sub steps may vary at subclass.
- **State**: encode complex state transitions.
- etc. (e.g. Interpreter)

Design Patterns: References

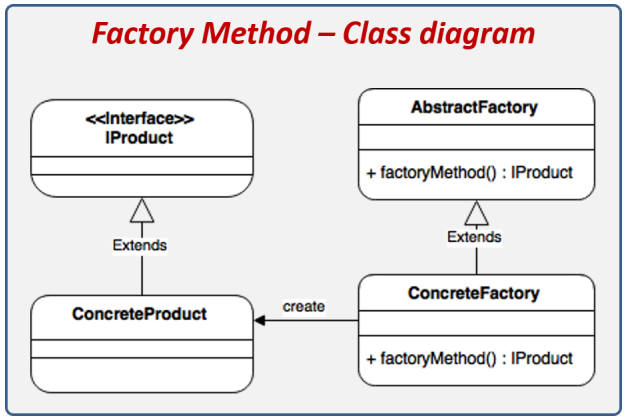
- Book: Head First Design Patterns
- SourceMaking: https://sourcemaking.com/design_patterns/
- ReactiveProgramming: https://reactiveprogramming.io/blog/en/design-patterns/factory-method
- Refactoring.Guru: https://refactoring.guru/design-patterns

Factory Method Pattern

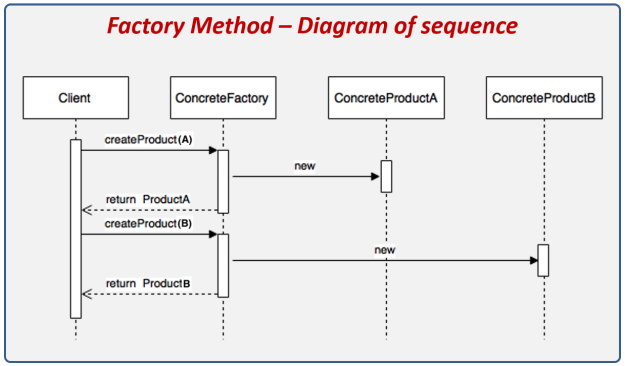


- Factory / Creator: include a factory method
- Concrete Factories / Concrete Creators: implement factory method
- Product
- Concrete Products

Factory Method: Class Diagram Draft

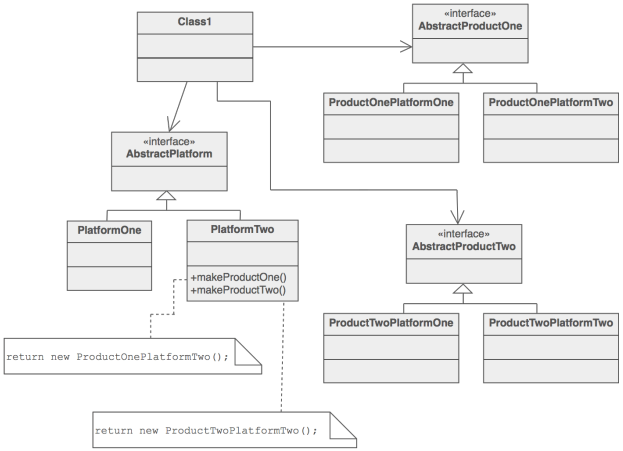


Factory Method: Sequence Diagram Draft



- **Not** an accurate Sequence Diagram.

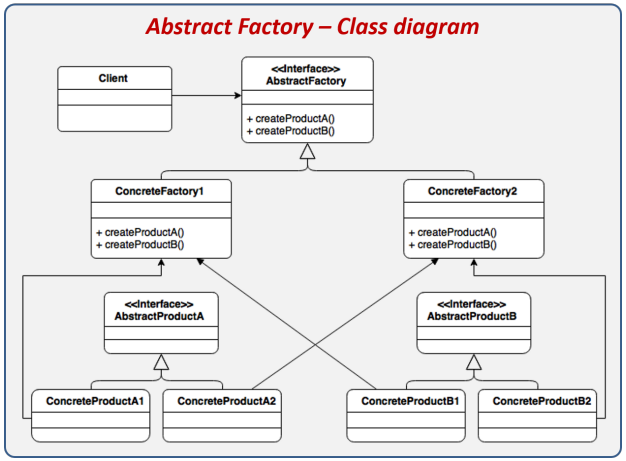
Abstract Factory Pattern



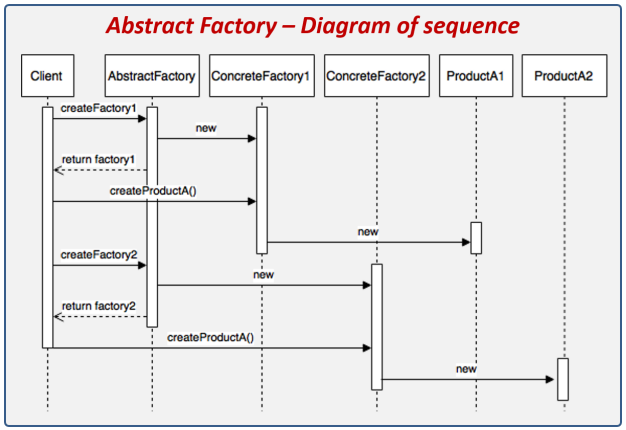
- Abstract Factory / Abstract Creator: include makeProductOne, makeProductTwo, etc.
- Concrete Factories / Concrete Creators: implement factory method
- ProductOne, ProductTwo, etc.
- Concrete ProductOneA, Concrete ProductOneB; Concrete ProductTwoA, etc.

When adding new products to the abstract factory, the interface has to be changed.

Abstract Factory: Class Diagram Draft

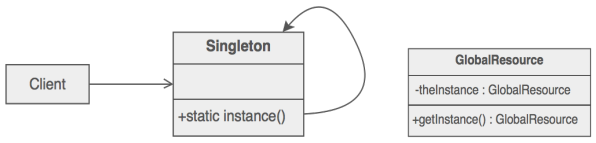


Abstract Factory: Sequence Diagram Draft



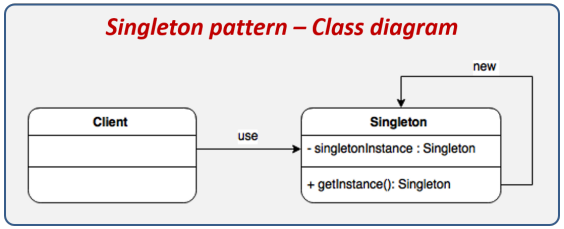
- **Not** an accurate Sequence Diagram.

Singleton Pattern

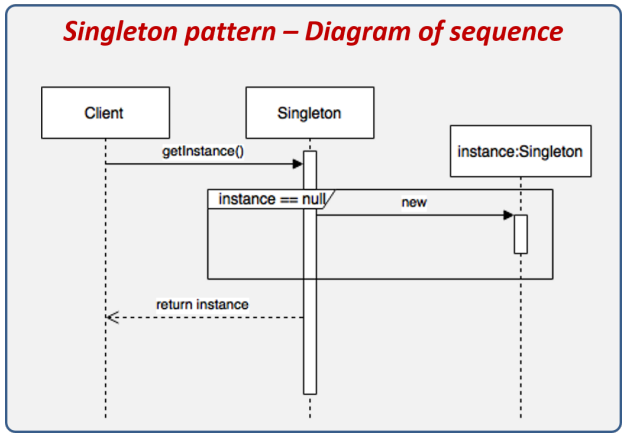


- The class of the single instance is responsible for access and "initialization on first use". The single instance is a private static attribute, accessed via a public static method.

Singleton: Class Diagram Draft

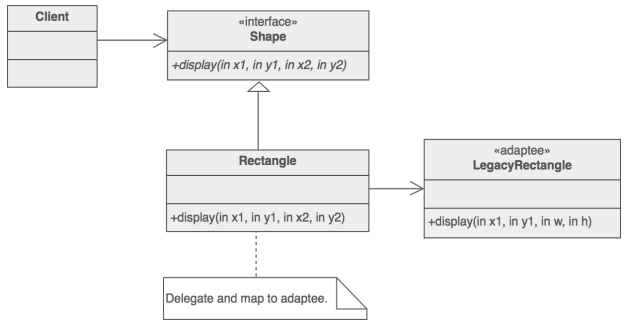


Singleton: Sequence Diagram Draft



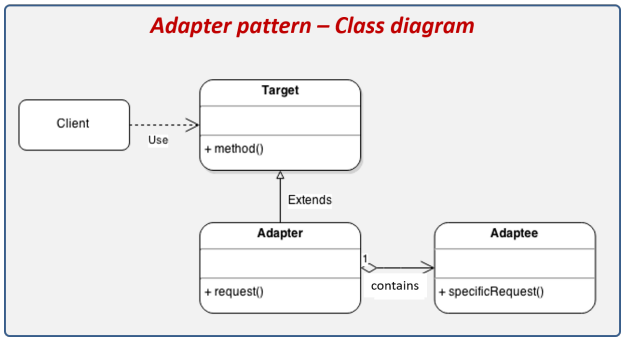
- **Not** an accurate Sequence Diagram.

Adapter Pattern

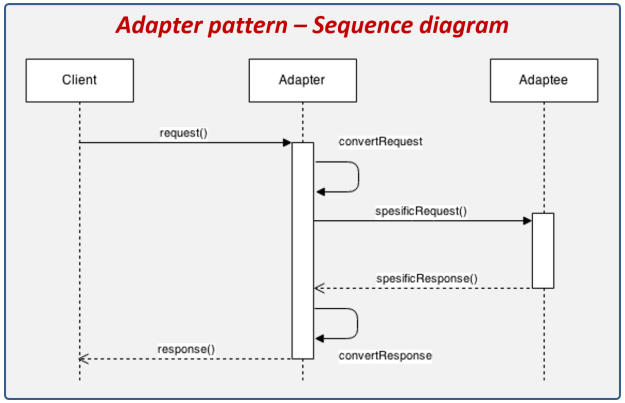


- Adapter: represents the implementation of the Target, hide details of Adaptee; e.g. Rectangle
- Adaptee: represents the class with the incompatible interface; e.g. LegacyRectangle
- Target: e.g. Shape

Adapter: Class Diagram Draft

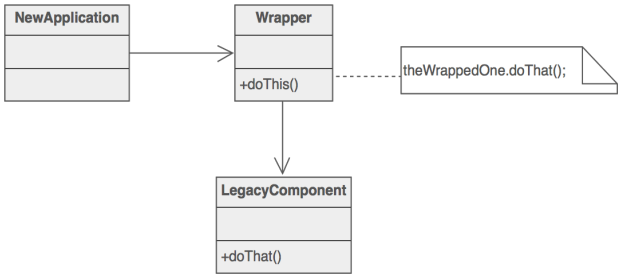


Adapter: Sequence Diagram Draft

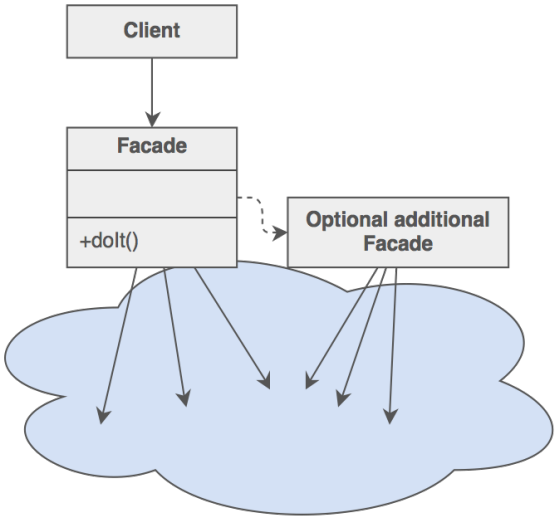


- **Not** an accurate Sequence Diagram.

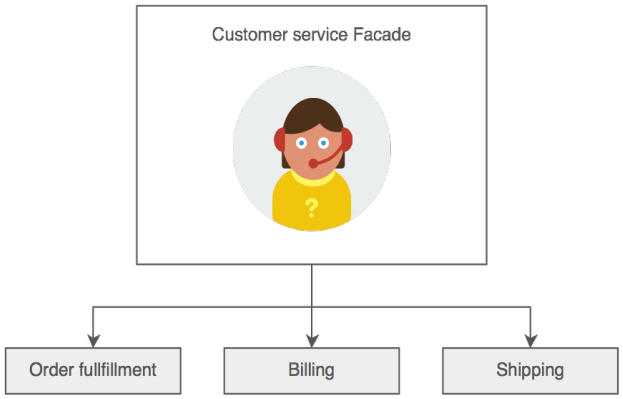
Adapter: a.k.a. Wrapper



Façade Pattern

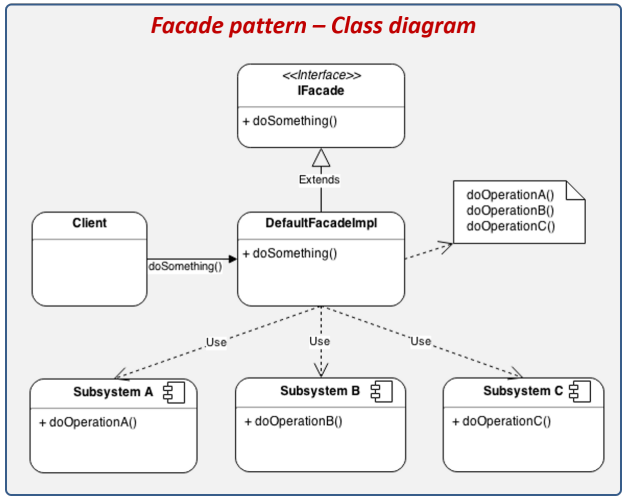


- The Façade defines a unified, higher level interface to a subsystem that makes it easier to use.
- IFacade: high-level interface, hiding the complexity of interacting with multiple systems.
- DefaultFacadeImpl: implementation of IFacade, in charge of communicating with all the subsystems.
- Subsystems: represents all the modules or subsystems with interfaces for communication.

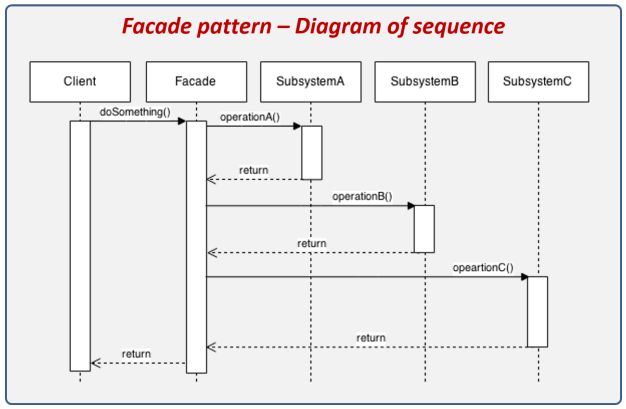


- As an example, the customer-service system could be incredibly complex without Façade.

Façade: Class Diagram Draft

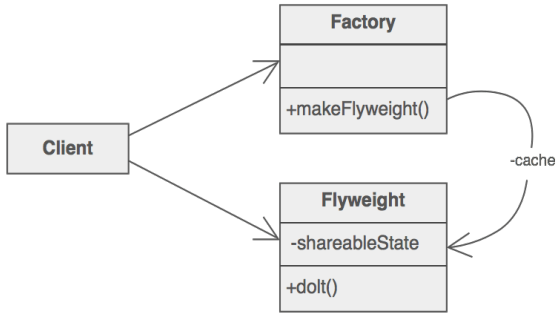


Façade: Sequence Diagram Draft



- **Not** an accurate Sequence Diagram.

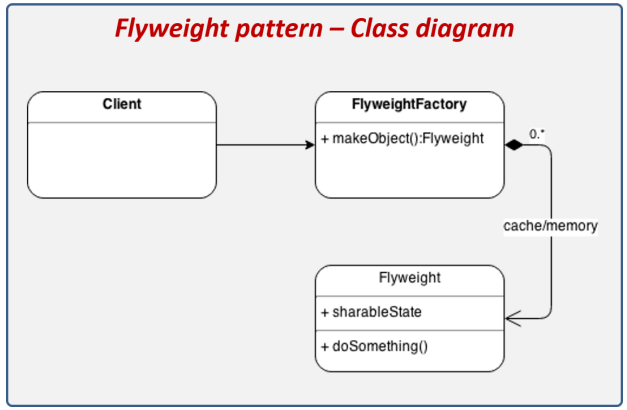
Flyweight Pattern



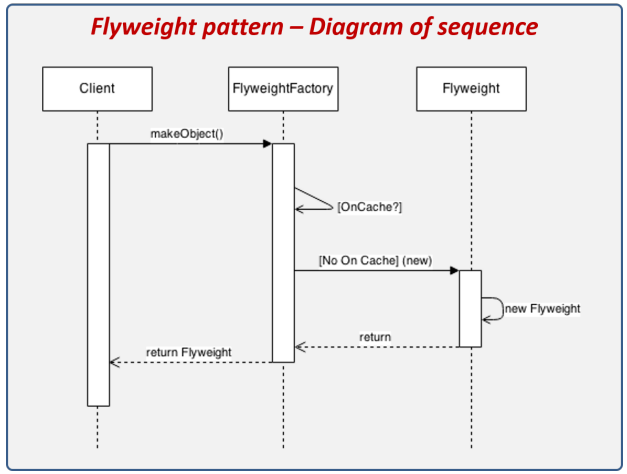
Flyweight Pattern

- FlyweightFactory: factory class for building the Flyweight objects.
- Flyweight: the objects we want to reuse in order to create lighter objects.

Flyweight: Class Diagram Draft

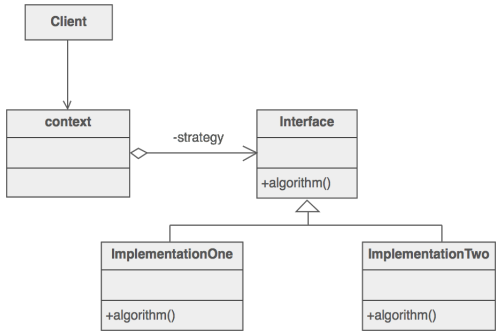


Flyweight: Sequence Diagram Draft



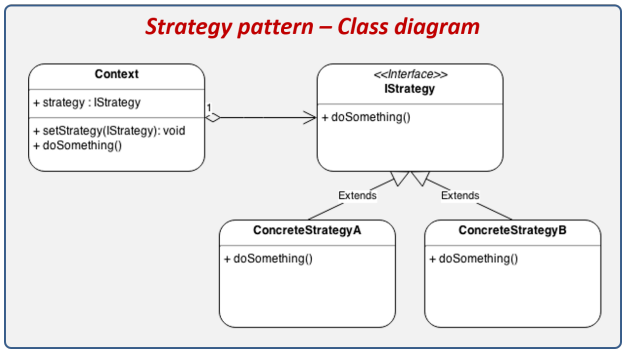
- **Not** an accurate Sequence Diagram.

Strategy Pattern

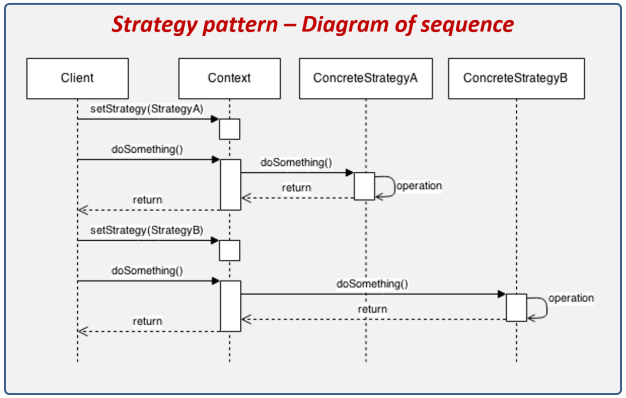


- Strategy Interface: define the common interface of all strategies that must implement.
- Concrete Strategy: inherit from Strategy Interface, they implement concrete strategies.

Strategy: Class Diagram Draft

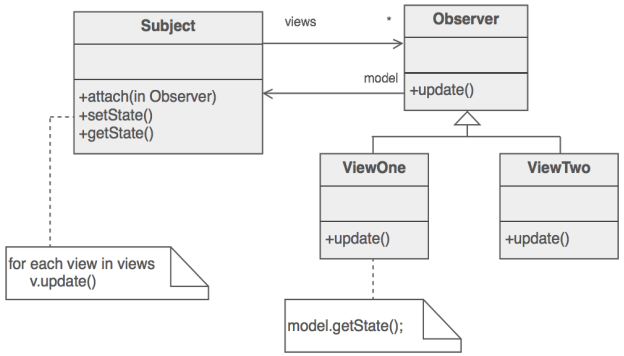


Strategy: Sequence Diagram Draft



- **Not** an accurate Sequence Diagram.

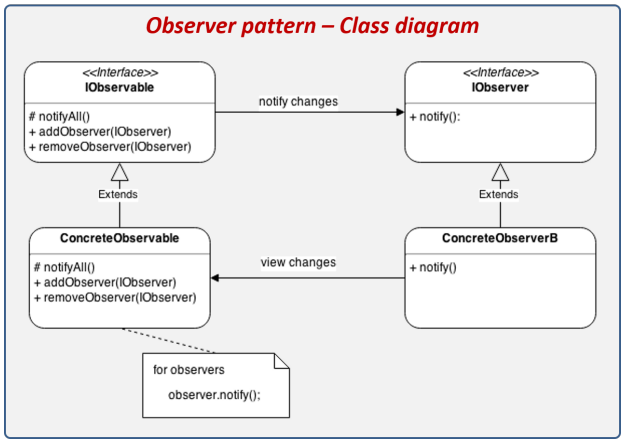
Observer Pattern



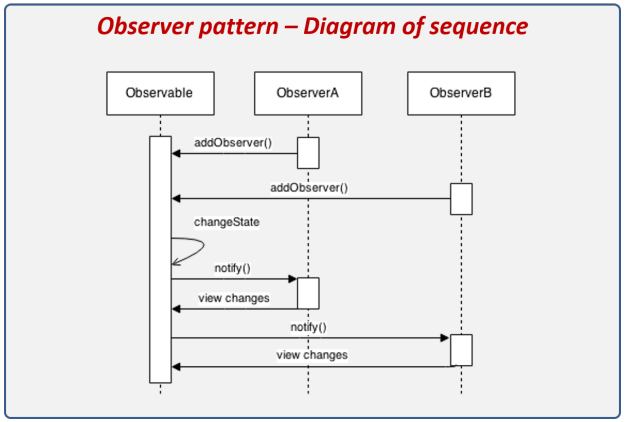
- Subject: interface of all observable subject classes, in it, methods that (1) keep track of observers listening to itself (2) notify the observers when change happens, are defined.
- Concrete Subject: the observable class; it implements all methods defined in Subject interface.
- Observer: interface observing the changes on Subject.
- Concrete Observer: Concrete class watching the changes on Subject, inherits from Observer, implements its methods.

It defines a one-to-many dependency between objects so that when one object (a concrete observable subject) changes state, all of its dependents (corresponding concrete observers) are notified and updated automatically.

Observer: Class Diagram Draft

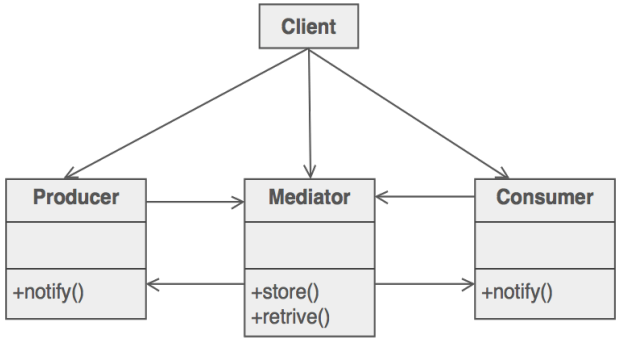


Observer: Sequence Diagram Draft



- **Not** an accurate Sequence Diagram.

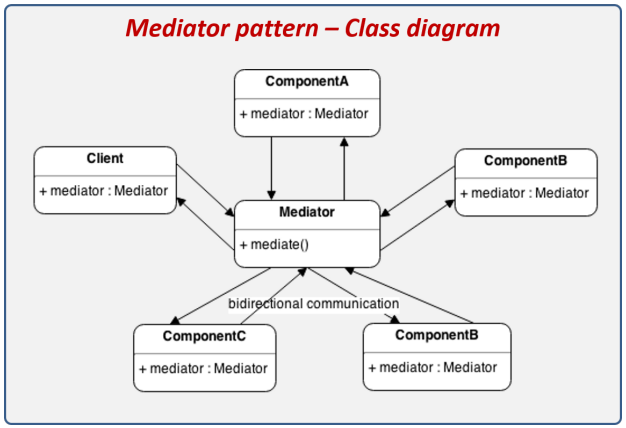
Mediator Pattern



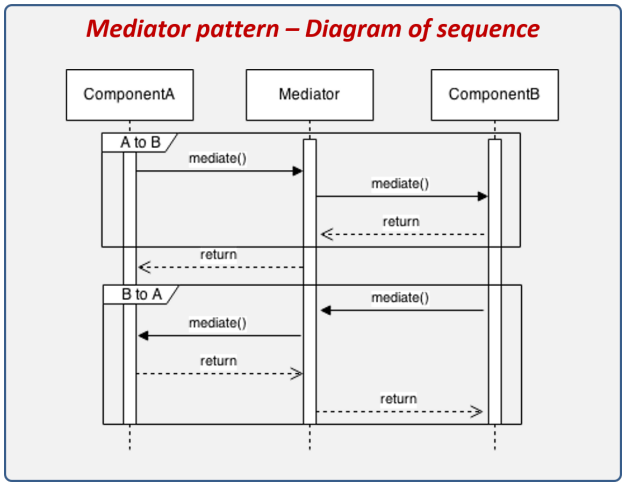
- Mediator: defines the interface for communication between colleague objects.
- Concrete Mediator: implements the mediator interface and coordinates communication between colleague objects.
- Colleague (Peer): defines the interface for communication with other colleagues
- Concrete Colleague: implements the colleague interface and communicates with other colleagues through its mediator only; e.g. Producer, Consumer in the figure.

Centralize many-to-many complex communications and control between related objects (colleagues).

Mediator: Class Diagram Draft

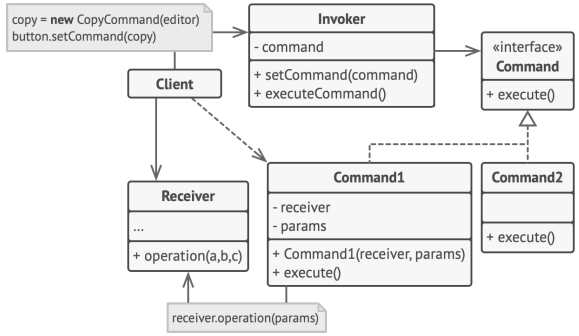


Mediator: Sequence Diagram Draft



- **Not** an accurate Sequence Diagram.

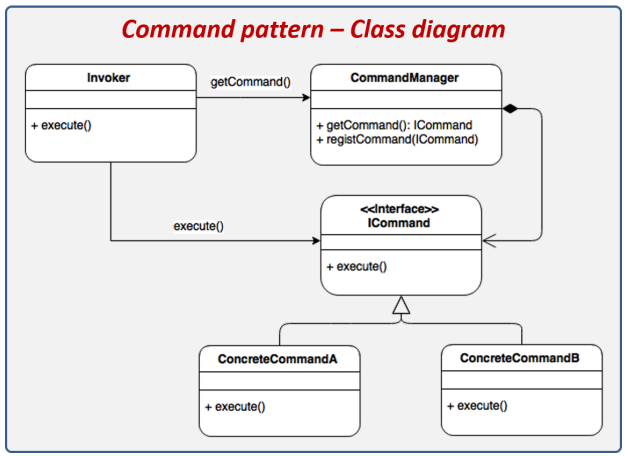
Command Pattern



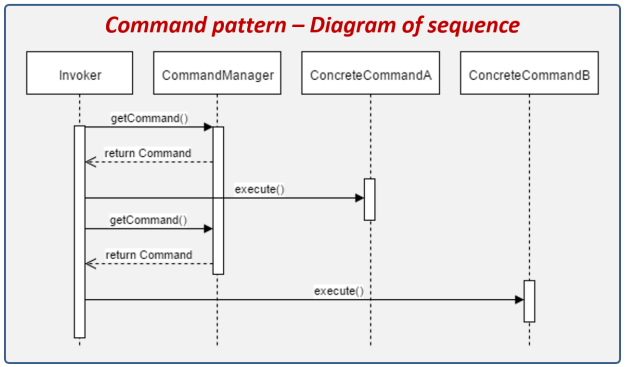
- Command: interface describing the structure of the commands, defining the generic execution method for all of them (e.g. execute, undo).
- Concrete Command: inheriting from Command, each of these classes represents a command that can be executed independently.
- Receiver: informed by the Concrete Command and take actions.
- Invoker: the action triggering one of the commands, hold a command and at some point execute it.
- (optional) Command Manager: manage all the commands available at runtime, from here we create / request commands.

The Command pattern allows requests to be encapsulated as objects, thereby allowing clients to be parameterized with different requests.

Command: Class Diagram Draft

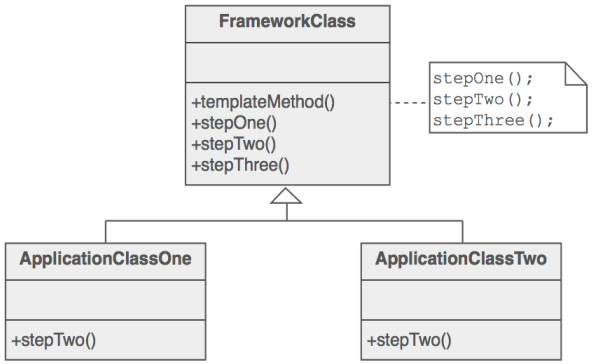


Command: Sequence Diagram Draft



- **Not** an accurate Sequence Diagram.

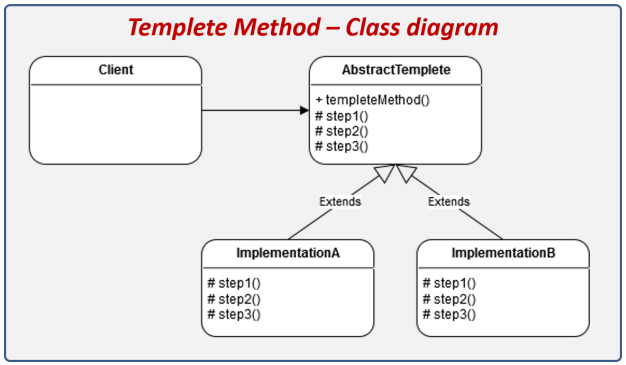
Template Method Pattern



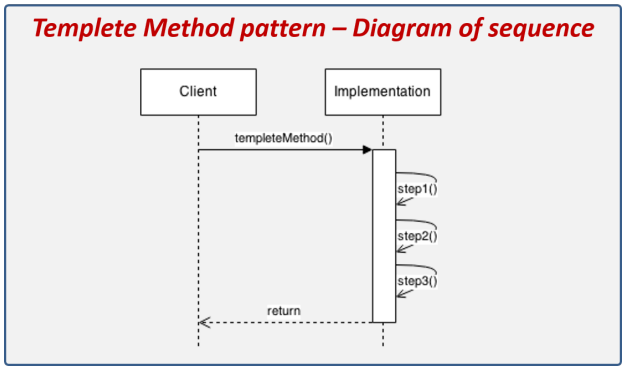
- Abstract Template: an abstract class including a series of operations which define the necessary steps for carrying out the execution of the algorithm; e.g. Framework Class in the figure.
- Implementation: the class inherits from Abstract Template and implements its methods to complete the algorithm; e.g. Application Class One / Two in the figure.

The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses; subclasses may redefine certain steps of an algorithm without changing its overall structure.

Template Method: Class Diagram Draft

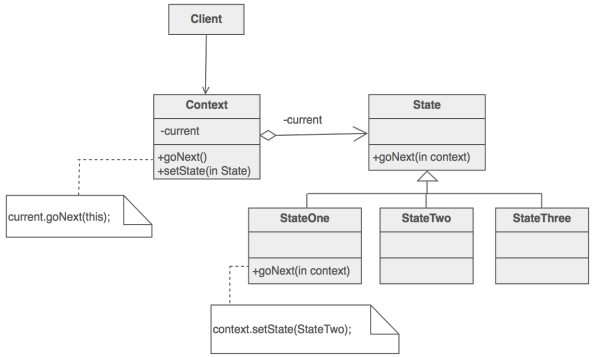


Template Method: Sequence Diagram Draft



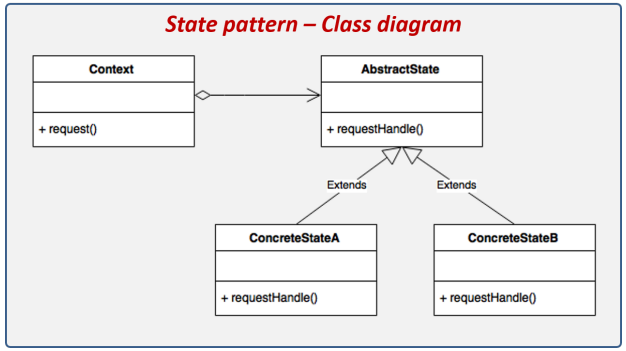
- **Not** an accurate Sequence Diagram.

State Pattern

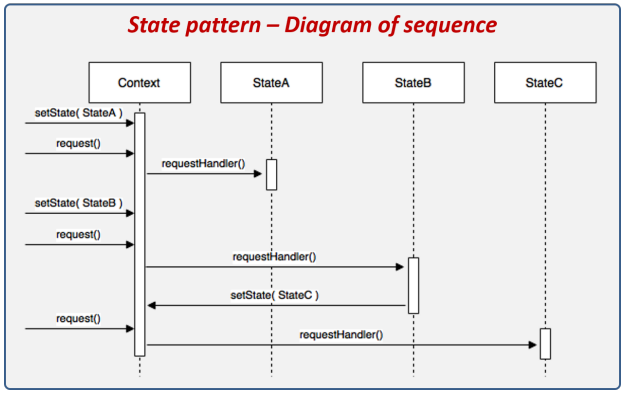


- Context: the component subject to changing states, it has its current state as one of its properties; e.g. in a vending machine example, this would represent the machine.
- State: abstract base class used for generating different states, usually works better as an abstract class, instead of as an interface, because it allows us to set default behaviors.
- Concrete State: inherit from State, each one of these represent a possible state the application could go through during its execution.

State: Class Diagram Draft



State: Sequence Diagram Draft



- **Not** an accurate Sequence Diagram.

Refactoring

- Semantic-preserving program transformations, made to the internal structure, without changing its observable behavior.
- Could involve changing implementation details, e.g. reorganization class hierarchies.
- Always guided by design patterns.
- Not formally defined, no way to check semantics preservation in practice. But it is a common vocabulary.
- Ref: `https://refactoring.guru/refactoring`

Fowler: Bad Code Smells

The symptoms of bad software design. Categories:

- **Bloaters** (complexity accumulates over time as the program evolves): long method, large class, **primitive obsession**, long parameter list, **data clumps**
- **Object-Orientation Abusers**: switch statements, temporary field, refused bequest, alternative classes with different interfaces
- **Change Preventers** (make change difficult): **divergent change**, **shotgun surgery**, **parallel inheritance hierarchies**
- **Dispensables** (pointless, unneeded parts): comments, duplicate code, **lazy class**, data class, dead code, **speculative generality**
- **Couplers** (causes excessive coupling): **feature envy**, **inappropriate intimacy**, message chains, middle man
- etc.

Primitive Obsession

Use of primitives instead of small objects for simple tasks (such as currency, ranges, special strings for phone numbers, etc.) e.g.:

- Use of constants for coding information;
- Use of string constants as field names for use in data arrays.

Possible solutions:

- replace data value with object;
- replace type code with class / subclasses.

Data Clumps

Bunches of data that hang around together, but ought to be made into their own object. Possible solutions:

- Extract class
- Introduce parameter objects
  - Replace a group of parameters that naturally go together with an object. e.g. start: Date, end: Date =¿ DateRange
- Preserve whole objects

Divergent Change v.s. Shotgun Survey

- **Divergent change**: one class, many kinds of changes;
- **Shotgun Survey**: one change, alters many classes.

Solutions: move method or move field, use inline classes, etc.

Parallel Inheritance Hierarchies

A special case of shotgun surgery: whenever you make a subclass of one class, you also have to make a subclass of another. Solutions: move method or move field.

Lazy Class

Class not useful enough to pay for the effort of maintenance should be eliminated. Possible solutions:

- lazy subclass: Collapse Hierarchy (merge the subclass to the superclass)
- more general: Inline Class (move all the class's features into another class and delete it)

Speculative Generality

Over-generated code preserving a lot of "will be useful in the future" components etc., containing unused class, method, field or parameter. Possible solutions:

- removing unused abstract classes: Collapse Hierarchy;
- unnecessary delegation of functionality: eliminated via Inline Class;
- methods with unused parameters: Remove Parameter;
- unused fields: simply deleted;
- methods with odd abstract names: Rename Method.

Feature Envy

A method accesses the data of another object more than its own data.

- e.g. a method that frequently invokes getter methods to another object to calculate some value. (usually data-related)

Inappropriate Intimacy

One class uses the internal fields and methods of another class. Possible Solutions:

- change bi-directional Association to unidirectional;
- in case of common interests, use Extract Class;
- Hide Delegate to let another class act as go-between.
  - A client class calling a delegate class's method (on the server) of an object. (e.g. client -¿ Person.getDept(), client -¿ Department.getManager(); =¿ client -¿ Person.getManager())

Replace Conditional with Polymorphism

An important **refactoring** technique.

- Problem: having a conditional that performs various actions depending on object type or properties.
- Solution: move each leg of the conditional into an overriding method in a subclass; make the original method abstract.

Benefits:

- instead of asking an object about its state and then performing actions based on this, it's much easier to simply tell the object what it needs to do and let it decide for itself how to do that (Tell-Don't-Ask principle)
- remove duplicate code, get rid of many almost-identical conditionals
- make it much easier to add a new execution variant, by add new subclass instead of changing conditions in existing code anywhere (Open/-Closed Principle)

Refactoring: Categories

- Data-Level Refactorings
- Statement-Level Refactorings
- Routine-Level Refactorings
- Class Implementation Refactorings
- Class Interface Refactorings
- System Level Refactorings

Refactoring: Safety

Problem:

- Potential of causing new bugs.

Solutions:

- Save the code you start with
- Keep refactorings small
- Do refactorings one at a time
- Make a list of steps you intend to take
- Make a parking lot— for changes that aren't needed immediately, make a "parking lot."
- Make frequent checkpoints
- Use your compiler warnings
- Retest
- Add test cases
- Review the changes
- Adjust your approach depending on the risk level of the refactoring
- etc.

A lot of research has been done in this field. (ref: lecture slides)

Testing

Overview:

- Testing and code review/inspection are the most common quality-assurance methods.
- Can **NEVER** guarantee bug-free.
- "Adversarial" role of of the rest of development activities, assuming you **WILL** find errors. (Developers will tends to skip more sophisticated kinds of test, and being overly optimistic.)
- Testing *by itself* **does NOT** improve software quality.

Categories:

- **Unit Testing**: execution of a complete class, routine, or small program, performed by the **development team**, **white box** testing of individual programs or executable modules.
- **Component Testing**: the execution of a class, package, small program, or other program element, performed by the **testing team**, **black box** testing on each part separately.
- **Integration Testing**: the combined execution of two or more classes / packages / components / subsystems etc.
- **System Testing**: the execution of the software in its final configuration, including integration with other software and hardware systems.
- **Regression Testing**: the repetition of previously executed test cases for the purpose of finding defects.

Another way of classification:

- **Black-Box Testing**: cannot see the inner workings of the item while being tested.
- **White-Box Testing**: the inner workings of the item being tested is visible to the tester.

Bounded Iteration and Infeasible Paths

- Counting the number of loop iterations for bounded programs;
- Identify infeasible paths through symbolic execution.

Symbolic execution Test Generation Regression test selection

Testing v.s. Debugging

- **Testing**: detecting / revealing errors.
- **Debugging**: solving the detected errors.

Test Adequacy Criteria

Necessity:

- We prefer the amount of test being just right (not too much, not insufficient).
- During software evolution, identifying relevant tests will help us save time.

Criterion:

- **Statement Coverage**: Is each statement (each line of code) executed?
- **Branch coverage**: Is every branch (if-else) evaluated on *both true and false* conditions?
- **Path coverage**: Is every possible path (note: each branch can have two paths) exercised by tests? (**much stricter** than the other two)

Control Flow Graph (CFG)

Similar to an **activity diagram**.
Diamonds for branch (outward lines should specify **conditions**), rectangles for statements (steps in the code), solid-line arrows representing "what's next".

Code Coverage Table

Often used with a **Control Flow Graph**.

Table Column	Explanation
Input	Exact assignment of input (e.g. { condition 1 = true, condition 2 = false, ... })
Exercised Statements	Statements reached under the input. (e.g. s1, s2, s5)
Exercised Branches	Branches selected under the current input. (e.g. b3, b8, b10)
Exercised Paths	(usually represented by a combination of the branches' selections, e.g. [b3, b8, b10])

Table's rows: each input row is followed by a **coverage** row, coverage is **accumulated** *from top to bottom*. It is very hard to guarantee 100% coverage for an arbitrary program.

Testing Tools in Practice

- **Unit Testing**: JUnit — automated framework for unit testing in Java, compare the observed proram state with the expected ones and reports the differences.
  - Each test case is realized by its own class derived from TestCase class (provided by JUnit).
  - Each test is realized by its own method whose name starts with *test...*
  - assertTrue() method inherited from TestCase means **assert**.
  - **Setting Up**: the method setUp() is called before each test of the class.
  - **Tearing Down**: the method tearDown() is called after each test. Useful for releasing fixture.
- **Component Testing**: **assertions**. Should be able to identify errors and stop execution immediately, reporting which test case is not passed. Test cases must be individually executed, independent from each other. Should be able to group tests into **test suites**.

Test suite: container for multiple test cases grouped together.

The Number of Paths

Consider each branch (if-else), if both paths are feasible, it results in 2 paths. With k branches, we have 2^k paths if no infeasible path.
In case of loop, we perform **loop unrolling** to count paths. If n loops on code with k branches, we have 2^{nk} paths if (1) there is no infeasible path, and (2) the later branches' executions are independent from earlier branches.
Possible questions being asked:

- #times a certain line is executed
- #branch executions (k) in the loop-unrolled program
- #path
- is there any infeasible path & which?
- draw control flow graph
- symbolic execution e.g. to design concrete input

Infeasible Paths / Dead Code

Cannot be executed under any inputs.
How to identify:

- Conduct a **symbolic execution**, to model individual paths, in terms of **logical constraints**.
- i.e. For each branch, examining whether there exists an input that makes the condition true / false.

Symbolic Execution

- Use fresh variables (1) in the beginning (2) after the state updates;
- Loop must be **unrolled** first;
- Propagate the constraints for both true and false evaluation of each branch;
- Conservatively estimating the effect of taking either path.

Symbolic Execution: Terms

Term	Explanation
Path Condition	The condition to exercise each path.
Effect	What happens if executing statements along one of the possible paths.

Test Input Generation

- Determine a path condition to exercise a particular path;
- Find concrete input assignments for each path using symbolic execution.

Equivalence Partitioning

The concept helps reduce the number of test cases required, formalizing the idea of

> "A good test case covers a large part of the possible input data."

(Edge) Coverage Matrix

Columns: Edge, Tests on Edge.
Edges are the links in a **control flow graph**, represented by the starting node and the ending node this time.
Tests on Edges indicates the **ids** of the tests conducted that has covered the corresponding edge.
If multiple edges are covered by exactly the same set of tests, we can merge the rows.

Regression Testing (Retesting)

Happens when code is changed (say, P =¿ P'), aims to verify whether or not the software hasn't "taken a step backward" / "regressed" (= not introducing known bugs).
Also suppose that T is a test suite for P, and P has passed all of the tests. The corresponding **coverage matrix** of T is C.

Regression Test Selection (RTS)

Given: (1) the difference between P and P' (2) C
Problem: identify a subset of T that can identify all regression faults. (Safe RTS)
Solution: Harrold & Rothermel's Regression Test Selection

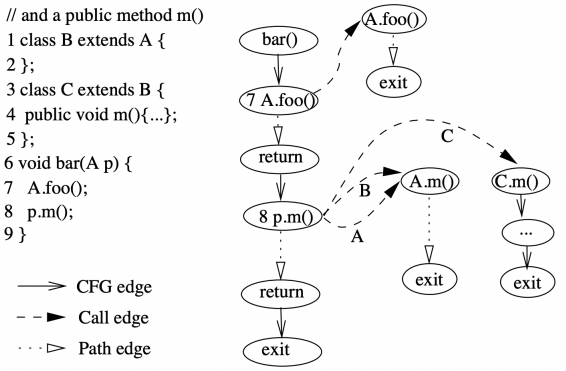
- A safe, efficient regression test selection technique
- Selection based on *traversal of control flow graphs (CFG)* for P and P'.
- Key Idea: select tests exercising **dangerous edges** in P'.
  - Dangerous Edges: the edges in CFG(P) where target node is different in CFG(P'); discovered by *traversing CFG(P) and CFG(P') in parallel*. Edges whose nodes are changed / removed / new...
  - Running all test cases in T that exercised dangerous edges is as effective as running entire T.

Key challenge in Java RTS: (1) polymorphism, (2) dynamic binding, (3) exception handling, and also handling external libraries & components.

- Solution: **JIG** (Java Interclass Graph) & adding *dynamic dispatching*

CFG with Dynamic Dispatching: Example

```
// A is externally defined
// and has a public static method foo()
// and a public method m()
1 class B extends A {
2 };
3 class C extends B {
4  public void m(){...};
5 };
6 void bar(A p) {
7   A.foo();
8   p.m();
9 }
```



Related Concepts

Prioritization: prioritizing and scheduling test cases. Provides assistance to regression testing.
Augmentation Methods: e.g. in class we introduced JIG and the CFG with dynamic dispatching to make RTS work in Java.
Change Impact Analysis: which tests are affected by program changes?

Random Testing v.s. Systematic Testing

Both used in real life.
Random Testing: randomly sample a huge amount of test cases inputs.
Systematic Testing: setup is defined, inputs are sampled systematically.

Modern Systematic Testing

Nowadays, problems are complex, forcing a hybrid:

- systematic in exploring search space;
- randomize to explore variation.

And also, prioritize **boundary conditions** first. e.g. null array, accessing an out-of-bound range.
White-Box Structural Testing is preferred: testing base on the structure of the program.

Structural Testing: Categories

Control oriented: how much of the control aspect of the code has been explored?

- e.g. statement coverage, branch coverage, path coverage (learned in class)

Data oriented: how much of the definition / use relationship between data elements has been explored?

Model-Based Testing

Black-Box Testing, based on some abstract test suits. **Model** refers to some abstract knowledge of the behavior of the system (not knowing the exact code structure). e.g. models could be:

- Decision trees / graphs
- Workflows e.g. for UML **Activity Diagrams** (node: activity, edge: state; similar yet different to state diagrams), adequacy criteria include:
  - Node coverage: cover all the nodes (activities);
  - Branch coverage: explore all directions at all decision nodes;
  - Mutations: what if the user does not follow the activity diagram?
- **Finite State Machines**: Good at describing interactions in systems with a small number of modes. e.g. adequacy criteria include:
  - Single State Path Coverage: collection of paths that cover each single state;
  - Single Transition Path Coverage: collection of paths that cover each single state / each single transition;
  - Boundary Interior Loop Coverage: criterion on number of times loops are exercised.
  - Mutation: how the system responds to unexpected inputs.
  - We can use probabilistic automata to represent distributions of inputs if we want to do randomized testing.
- Grammars: used to describe well-formed inputs to systems, used to construct sample inputs. Use coverage criteria on a test set to see that all constructs are covered.
- etc.

Mutation Testing

White-Box, **Structural** Testing method, aimed at *assessing / improving the adequacy of test suites*, and *estimating the number of faults present in systems* under test. Proved to be able to effectively emulate real world faults in *Is mutation an appropriate tool for testing experiments?* (ICSE'05, Andrews J.H et al.). Major steps:

1. systematically apply mutations to P, resulting in a sequence $\{P_1, P_2, \ldots, P_n\}$, consisting of mutants of P. P_i is derived by applying a single mutation operation to P. P_i is expected to executable but buggy.
2. Run T on all P_i, where T will **kill** P_i if it detects and error.
3. If we kill k out of n mutants, the adequacy of T is measured by the quotient $\frac{k}{n}$, which is called a *mutant killing ratio*. T is *mutation adequate* if $k = n$.

Kinds of mutations here:

- **Value Mutations**: changing the values of constants or parameters (e.g. loop bounds — being one out on the start or finish)
- **Decision Mutations**: modifying conditions (e.g. at a branch — changing greater than to less than)
- **Statement Mutations**: might involve deleting / duplicating certain lines or permuting the order of lines of code etc. (e.g. in a loop — omit the increment on some variable) Other possibilities include: changing operations in arithmetic expressions.

Hoare Logic: Introduction

Hoare-style Program Verifications, named after Sir Tony Hoare (British), also known as Hoare logic, is widely used in communicating Sequential Processes. In general, Hoare Logic:

- is a formal system with a set of logical rules for reasoning rigorously about the correctness of computer programs;
- is a formal inspection technique, good for code review & pair programming, conveys pre/post-conditions well.

Hoare Logic: Hoare Triples

The central feature of Hoare logic is the Hoare triple:

$$\{P\}S\{Q\},$$

where P serves as pre-condition, Q is the post-condition, and S is the corresponding piece of program. S is **executed** if the predicate P is true, then **terminated** if the other predicate Q is true.

Hoare Triples: Weakest Precondition (wp)

$\text{wp}(S, Q) = P$ is the most **general** P such that

$$\{P\}S\{Q\}$$

Note: Here is how we represent *replacing w by E in R*,

$$\text{wp}(w := E, R) \equiv R[w := E]$$

e.g.

$$\begin{aligned}\text{wp}(x := x + 1, x \leq 10) &\equiv \{(x \leq 10)[x := x + 1]\} \\ &= \{x + 1 \leq 10\} = \{x \leq 9\}\end{aligned}$$

$$\{P\}S\{Q\} \iff \big(P \Rightarrow \text{wp}(S, Q)\big)$$

$$\text{wp}(S; T, R) \equiv \text{wp}\big(S, \text{wp}(T, R)\big)$$

$$\begin{aligned}&\text{wp}(\text{if } B \text{ then } S \text{ else } T \text{ end}, R) \\ \equiv&\big(B \Rightarrow \text{wp}(S, R)\big) \wedge \big(\neg B \Rightarrow \text{wp}(T, R)\big) \\ \equiv&\big(B \wedge \text{wp}(S, R)\big) \vee \big(\neg B \wedge \text{wp}(T, R)\big)\end{aligned}$$

$$\text{wp}(\text{assert } P, R) \equiv P \wedge R$$

Hoare Triples: Example

$$\begin{aligned}\{P\}S\{Q\} \wedge \{P\}S\{R\} &\Rightarrow \{P\}S\{Q \wedge R\} \\ \{P\}S\{Q\} \wedge \{Q\}S\{R\} &\Rightarrow \{P \vee Q\}S\{R\} \\ \{P\}S\{Q\} \wedge \{Q\}T\{R\} &\Rightarrow \{P\}S; T\{R\} \\ \{P \wedge B\}S\{R\} \wedge \{P \wedge \neg B\}T\{R\} &\Rightarrow \{P\} \text{ if } B \text{ then } \\ & S \text{ else } T \text{ endif } \{R\}\end{aligned}$$

If S is a loop body and P is the loop invariant (after loop it still holds):

$$\{P \wedge B\}S\{P\} \Rightarrow \{P\} \text{ while } B \text{ do } S \text{ done } \{\neg B \wedge P\}$$