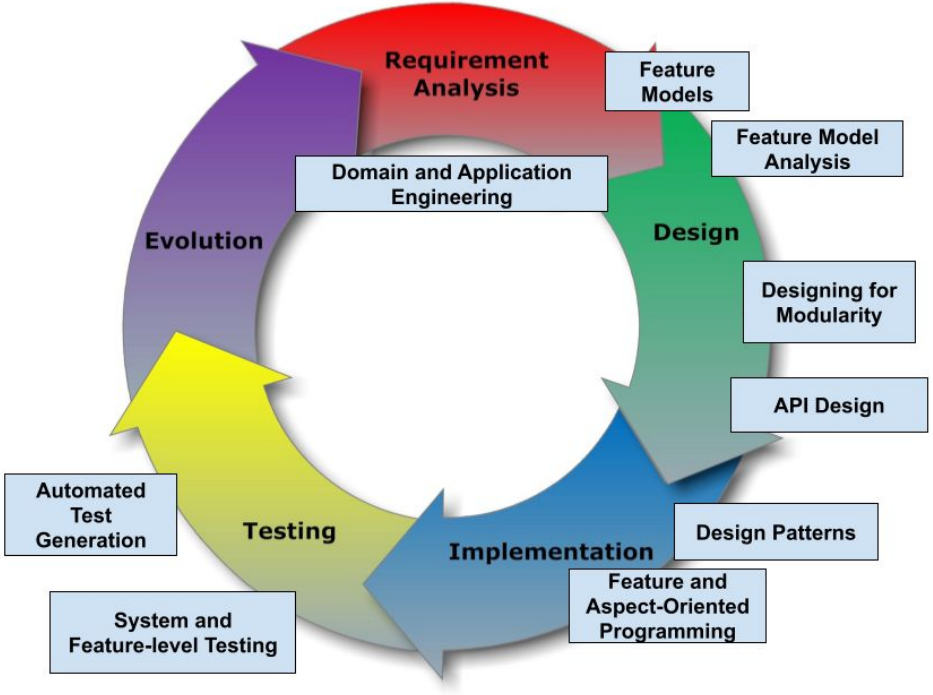
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Lecture 14: Course Summary

Gregory Gay and Daniel Strüber
TDA 594/DIT 593 - December 15, 2022

SE Principles for Complex Systems



Complex???????????

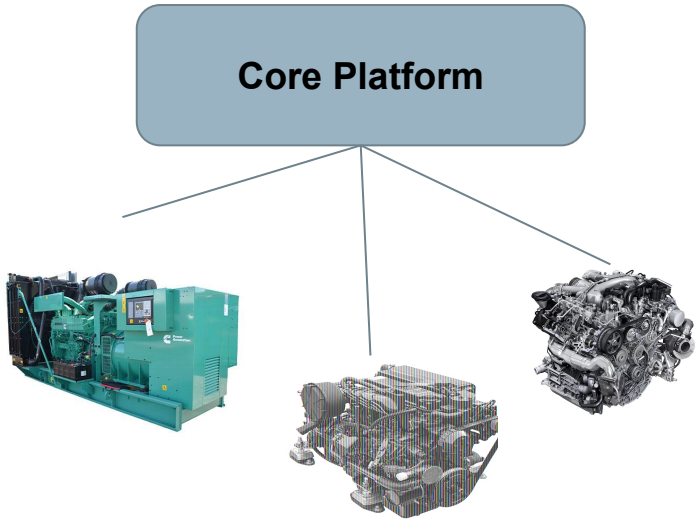
- **Variability**
  - The ability to change and customize software to deliver **variants** to new users.
  - Requires designing code **to be reused** and to **work with reused code**.
- **Changeability and Maintainability**
  - The ability to **add new features and options** while ensuring that **existing code still works**.

Why Change?

- The Law of Continuing Change
  - A program used in a real-world environment must change, or become progressively less useful in that environment.
- The Law of Increasing Complexity
  - As a program evolves, it becomes more complex.
  - Resources are needed to preserve and simplify structure

Software Product Lines (SPLs)

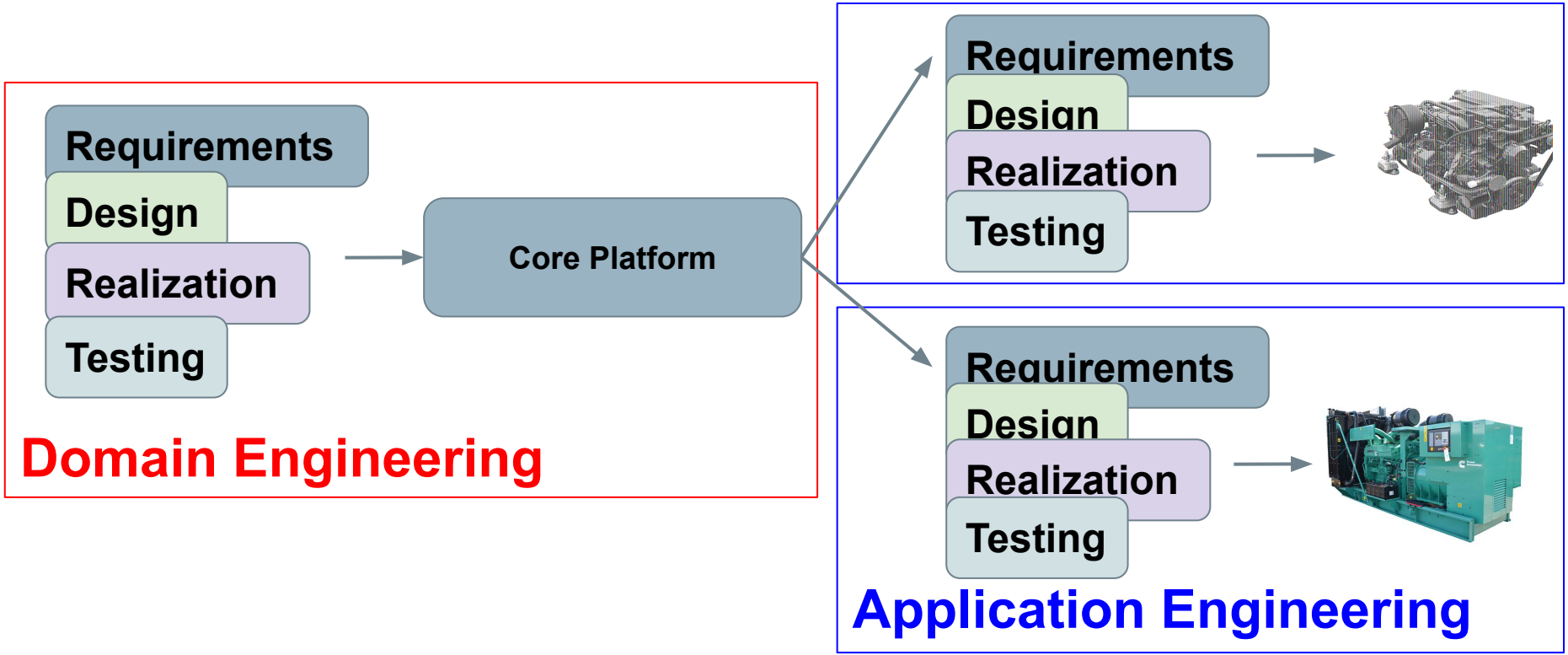
- Highly configurable families of systems.
- Built around common, modularized features.
  - Common set of core assets.
- Allows efficient development, customization.



Why Software Product Lines?

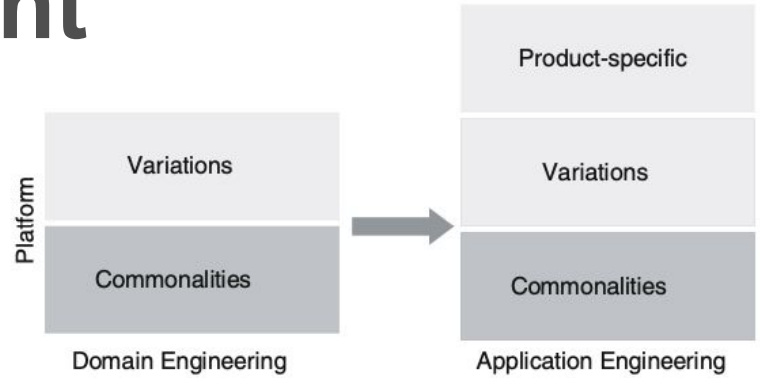
- Designed **FOR** reuse of assets.
- Designed **TO** reuse assets.
- Successful SPLs are **highly configurable**, and **evolve easily** over time.
- Most modern systems intend to achieve at least one of these two tasks.
  - **SPLs achieve both.**

Domain and Application Engineering



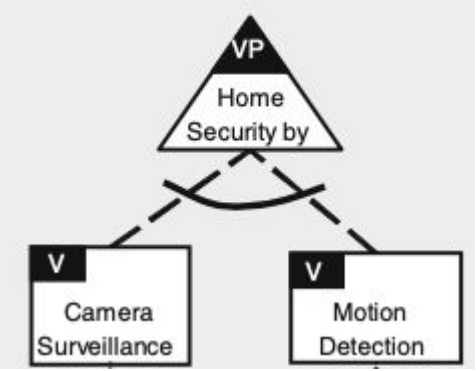
Variability Management

- Commonality
  - Shared between all products.
  - Implemented in core platform.
- Variations
  - Shared by subset of products.
  - Implemented in core platform, enabled in subset.
- Product-specific
  - Unique to a single product.
  - Platform must support unique adaptations.



Reasoning about Variability

- **Variation Point**
  - Where one product can differ from another.
  - Ex: Which features are supported by this security alarm?



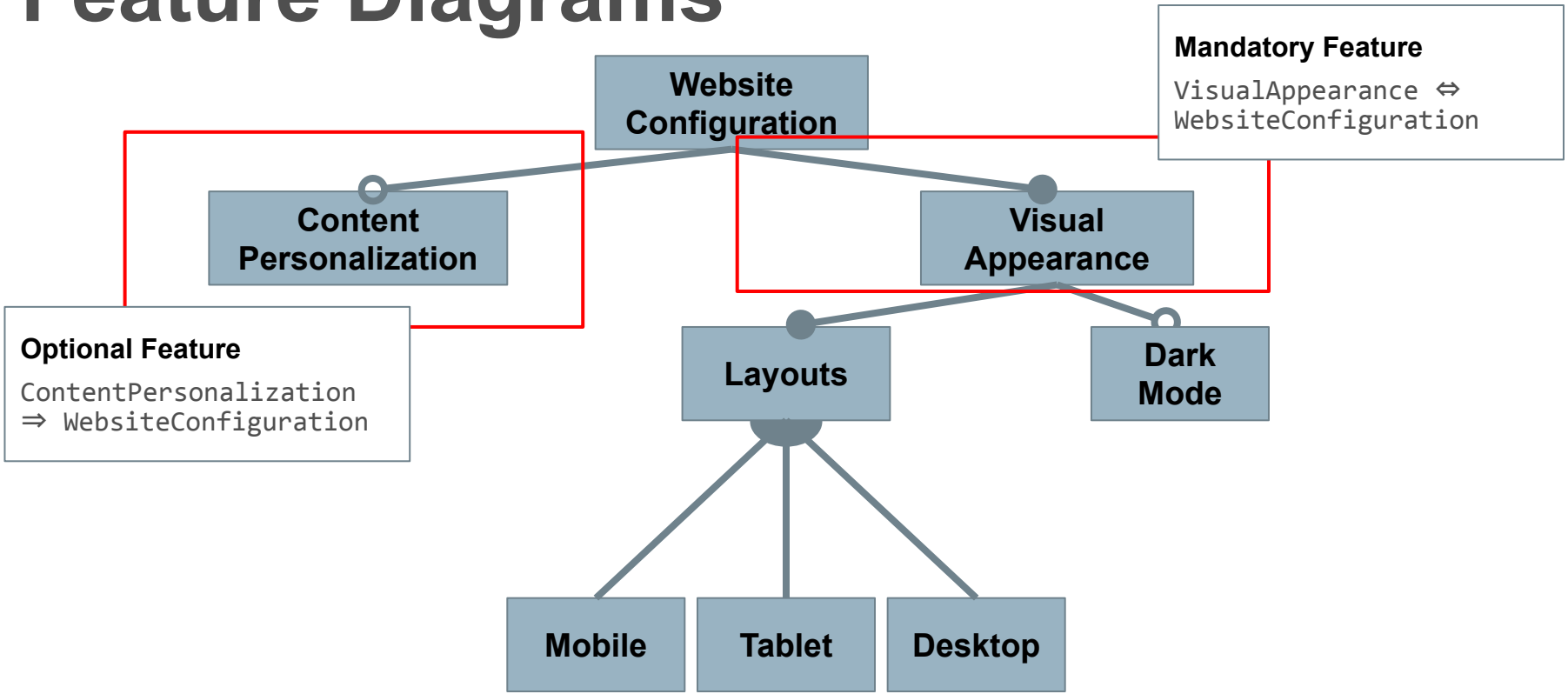
- **Feature**
  - Options that can be chosen at each variation point.
  - Ex: Motion detection, camera

Feature Diagrams

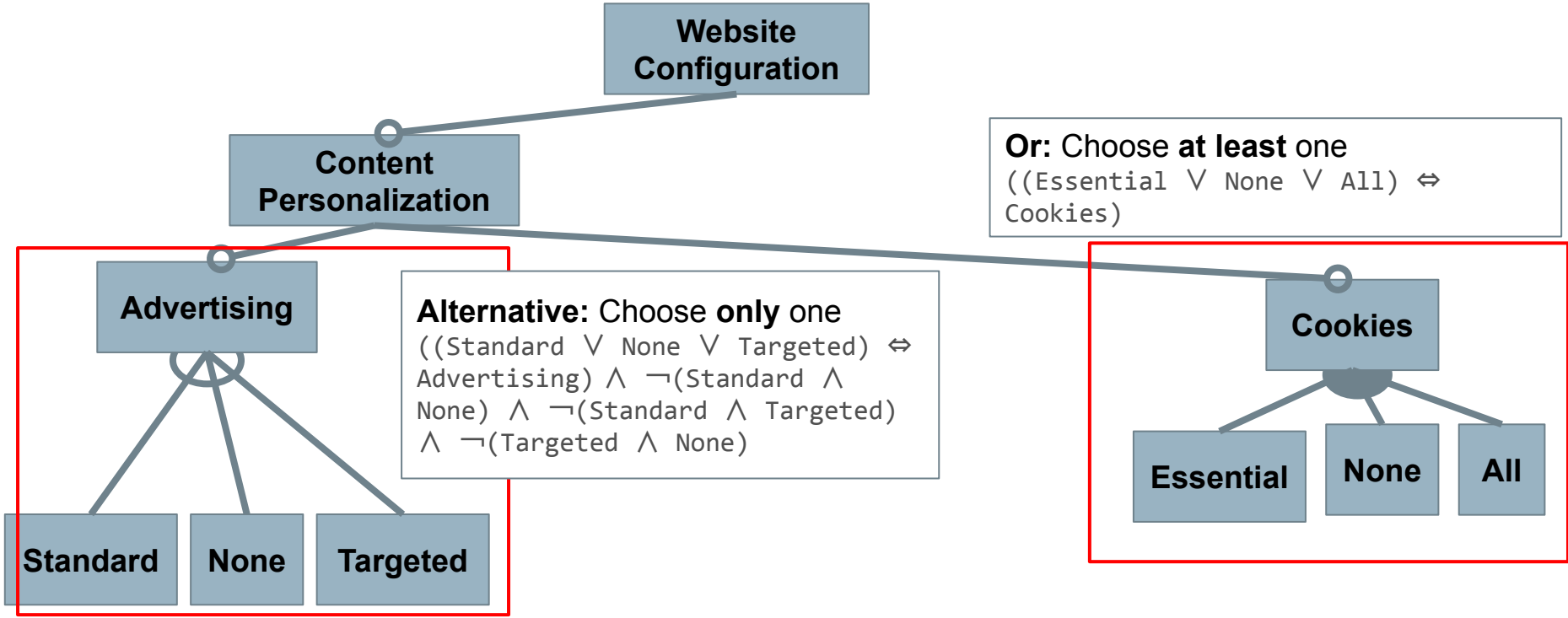
Feature Modeling

- A specification of variation points and features in a hierarchical form.
  - Represented visually using **feature diagrams**.
  - Also represented as **propositional logic** for analysis.
- Enables understanding of dependencies and what valid products can be built using a platform.

Feature Diagrams

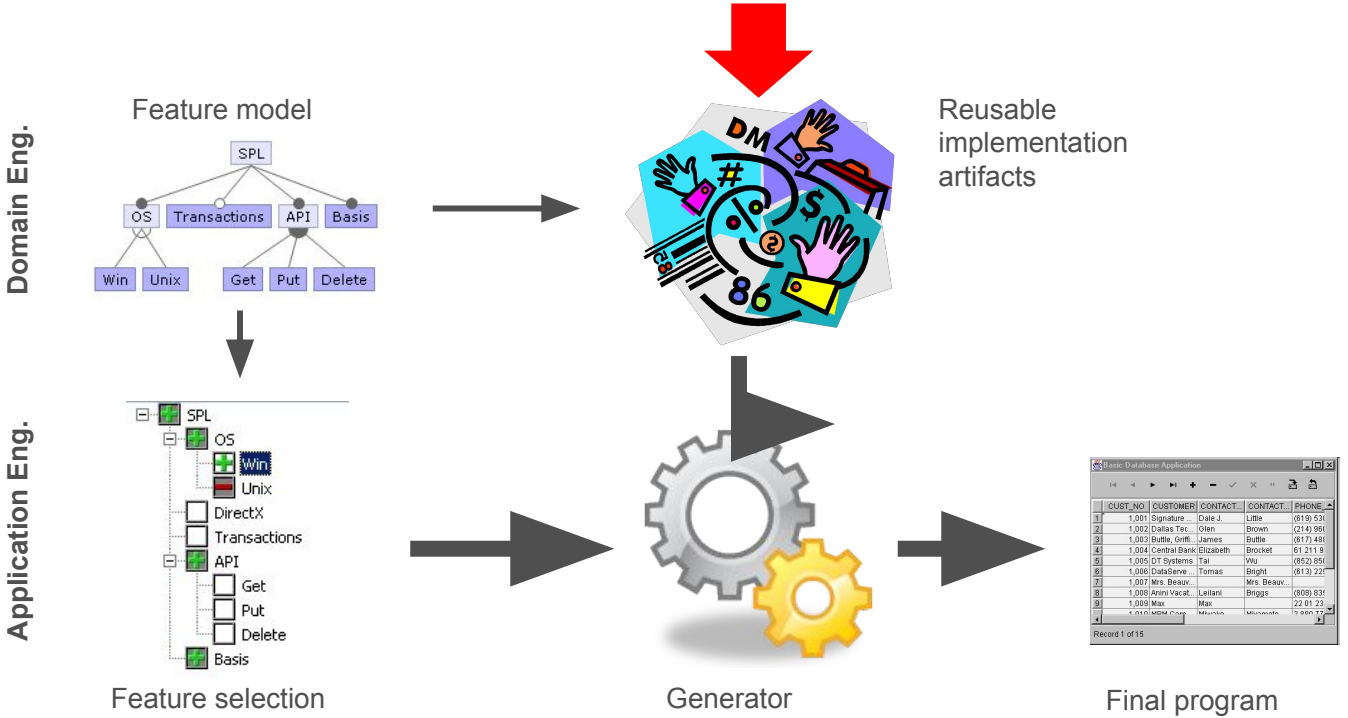


Example - Website Configuration



Implementation

How to implement variability?



Annotation-Based Representation

- Code in common code base.
- Code related to a feature is marked.
  - Preprocessor annotations, if-statements.
- Code belonging to deselected features:
  - ignored (load-time, run-time)
  - removed (compile-time).

Composition-based Representation

- Feature code in dedicated location.
  - Class, file, package, service
- Selected units combined to form product.
- Requires clear mapping between features and units

Variability with Preprocessors

- Selectively include or exclude code before compilation
- Developer wraps conditional code with preprocessor directives
- Preprocessor removes deselected code before compilation
- Exist for many languages, most famous: cpp (for C, C++)

```
1  class Node {
2      int id = 0;
3
4      //#ifdef NAME
5      private String name;
6      String getName() { return name; }
7      //#endif
8      //#ifdef NONAME
9      String getName() { return String.valueOf(id); }
10     //#endif
11
12     //#ifdef COLOR
13     Color color = new Color();
14     //#endif
15
16     void print() {
17         //#if defined(COLOR) && defined(NAME)
18         Color.setDisplayColor(color);
19         //#endif
20         System.out.print(getName());
21     }
22 }
23 //#ifdef COLOR
24 class Color {
25     static void setDisplayColor(Color c){/*...*/}
26 }
27 //#endif
```

Variability with Build Scripts

- Compiles code conditionally depending on features selected.
  - Feature selection read from file or inferred from environment (language, location, software).
  - Features can control how files compiled.

```
1 #!/bin/bash -e
2
3 rm *.class
4 javac Graph.java Edge.java Node.java \
5       Color.java
6 jar cvf graph.jar *.class
```

```
1  #!/bin/bash -e
2
3  if test "$1" = "--withColor"; then
4    cp Edge_withColor.java Edge.java
5    cp Node_withColor.java Node.java
6  else
7    cp Edge_withoutColor.java Edge.java
8    cp Node_withoutColor.java Node.java
9  fi
10
11 rm *.class
12 javac Graph.java Edge.java Node.java
13 if test "$1" = "--withColor"; then
14   javac Color.java
15 fi
16
17 jar cvf graph.jar *.class
```

Variability with Parameters

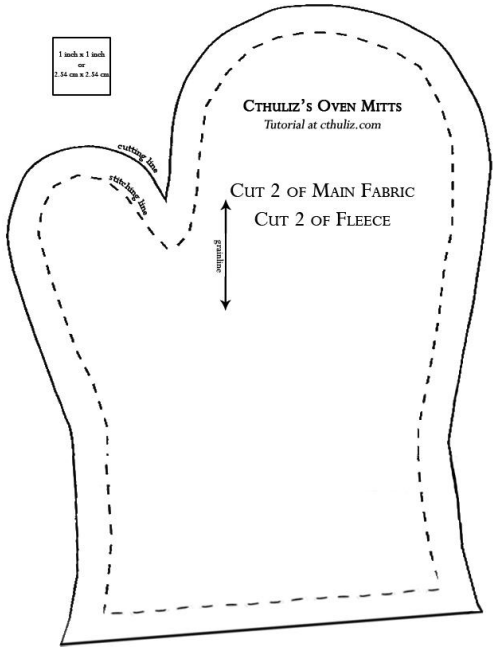
- Use conditional statements to alter control flow based on features selected.
- Boolean variable based on feature, set globally or passed directly to methods:
  - From command line or config file (load-time binding)
  - From GUI or API (run-time binding)
  - Hard-coded in program (compile-time binding)

Design Patterns

Design Patterns

Don’t just describe *classes*, describe ***problems***.

Patterns prescribe design guidelines for common problem types.

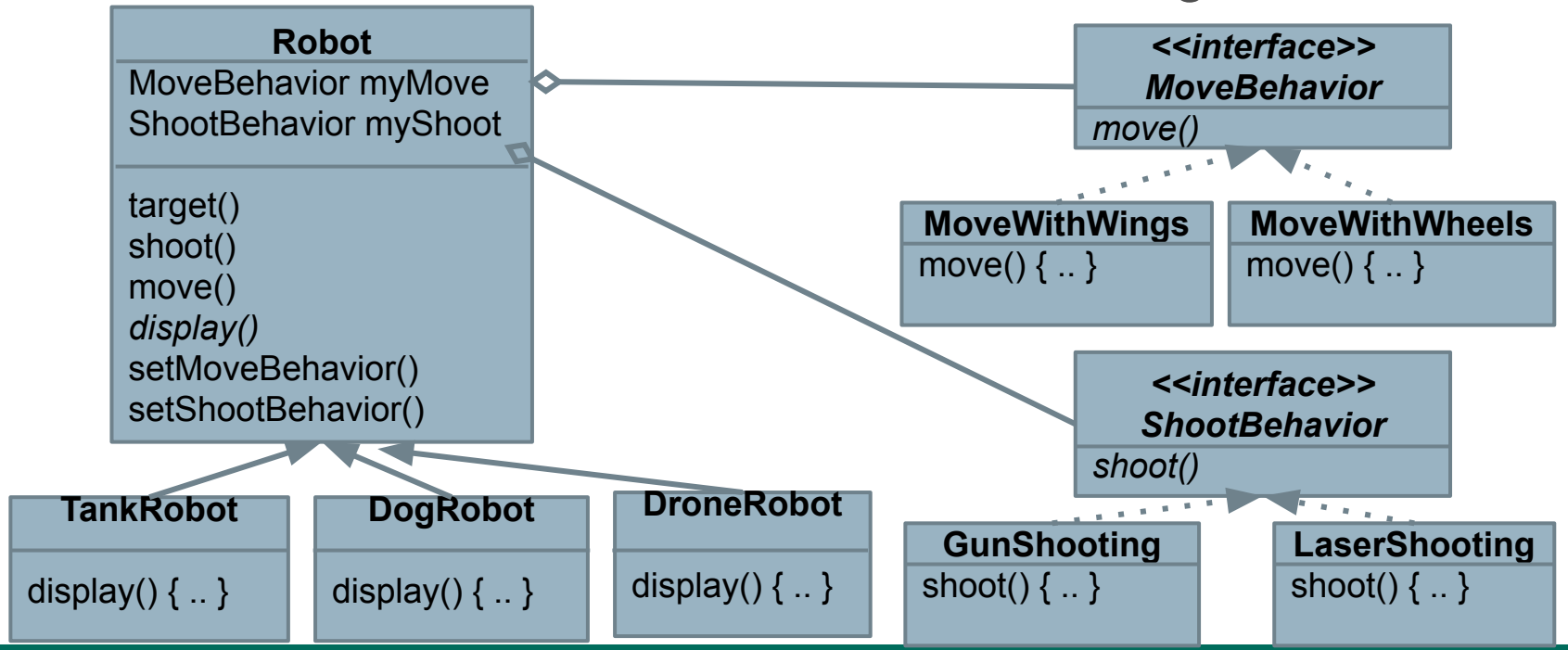


Principles of Design

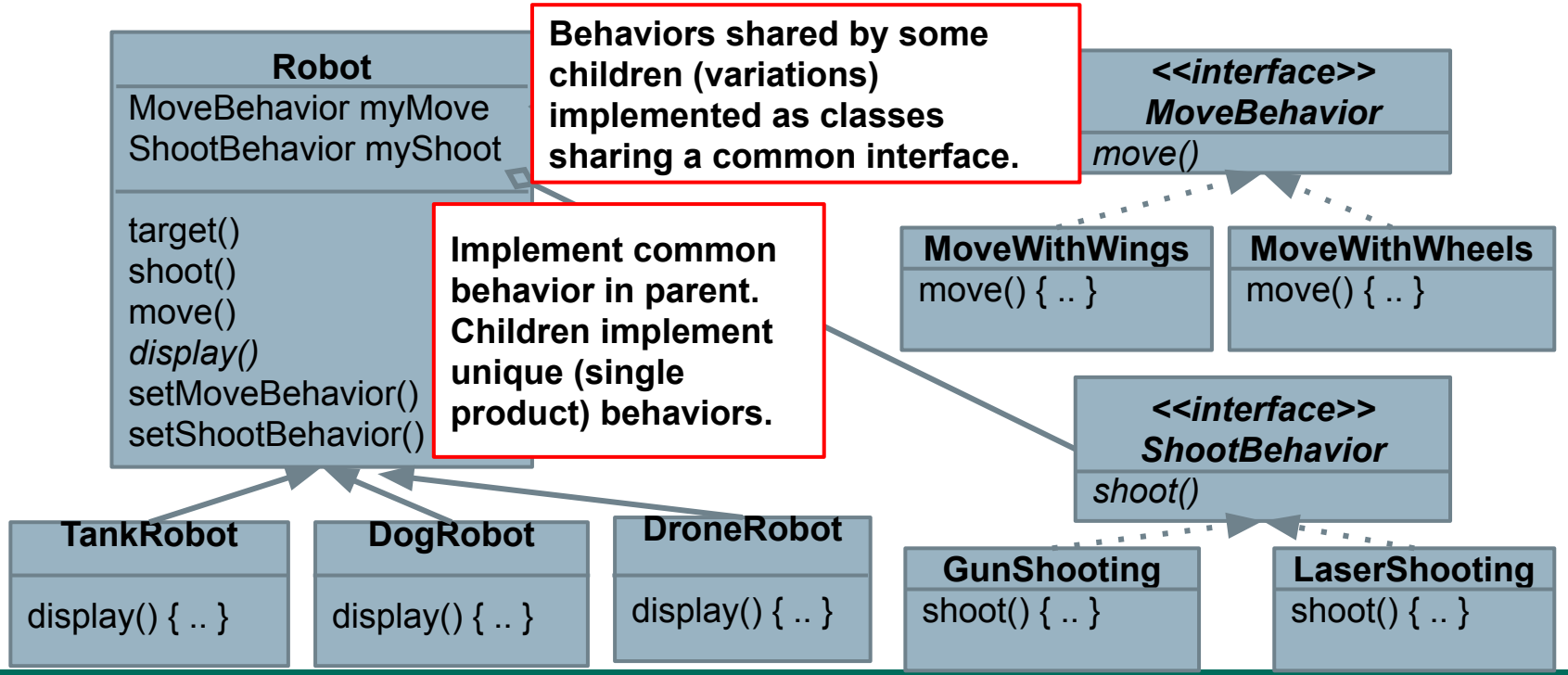
1. Identify aspects that vary and encapsulate them away from what doesn't.
2. Program to interface rather than implementation.
3. Favor composition over inheritance.
4. Open for extension, but closed for modification.
5. Talk only to your immediate friends.

Strategy Pattern

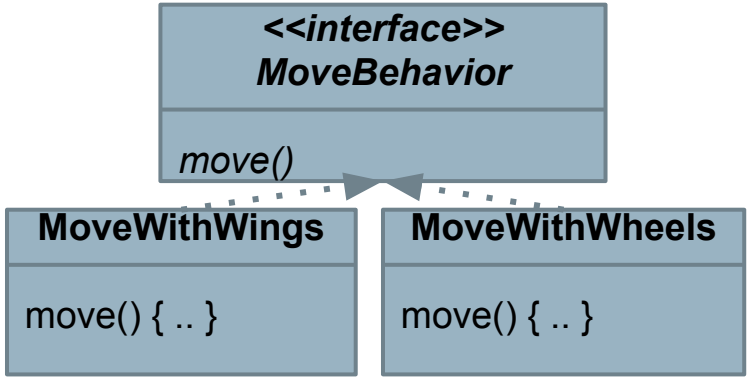
Defines family of algorithms, encapsulates them, makes them interchangeable.

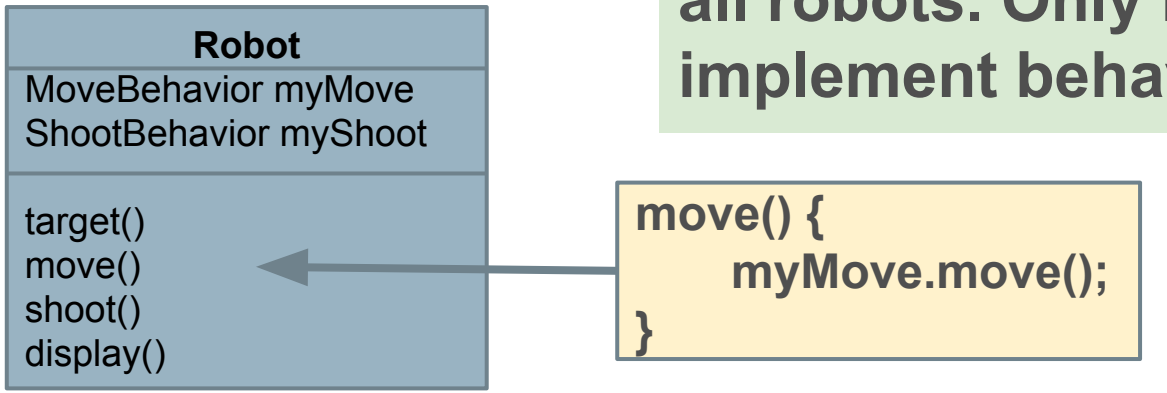


Identify aspects that vary and encapsulate them away from what doesn't



Program to interface rather than implementation



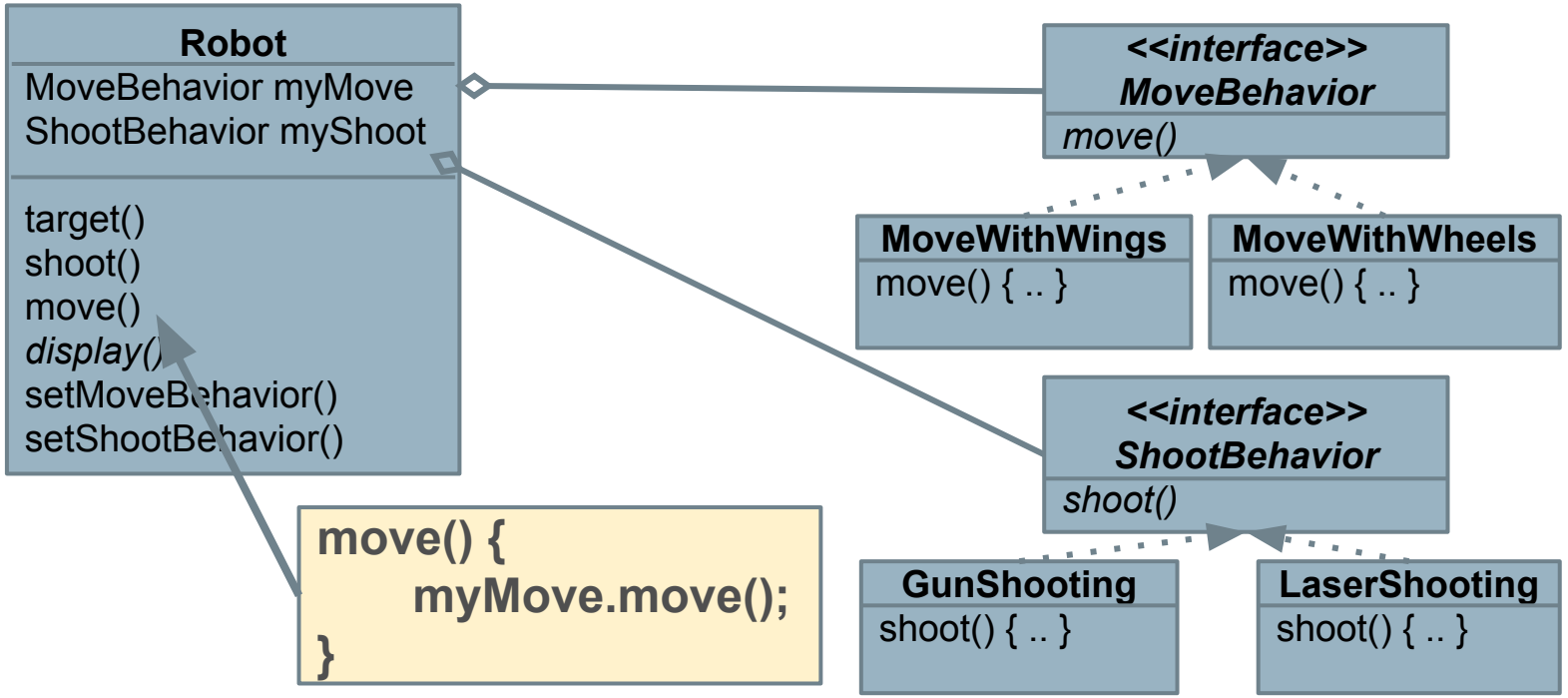



```
Robot d = new DroneRobot();
d.move();
```

Behavior called in same way for all robots. Only need to implement behavior once.

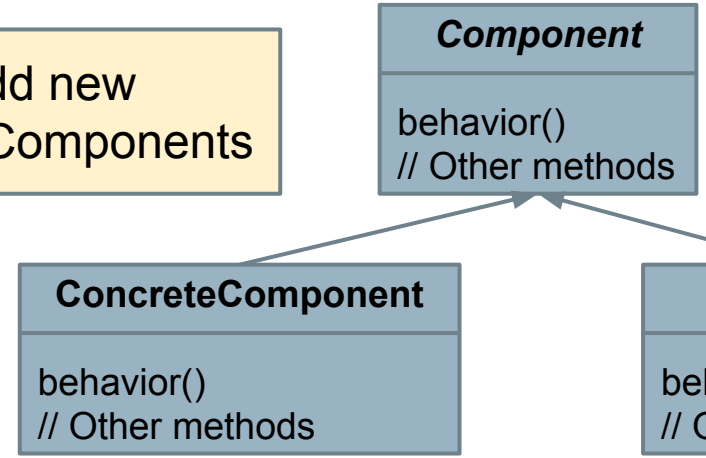
Favor composition over inheritance

A class can be composed of building blocks of encapsulated variable code. Class calls into the block.



The Decorator Pattern

Decorators add new behaviors to Components

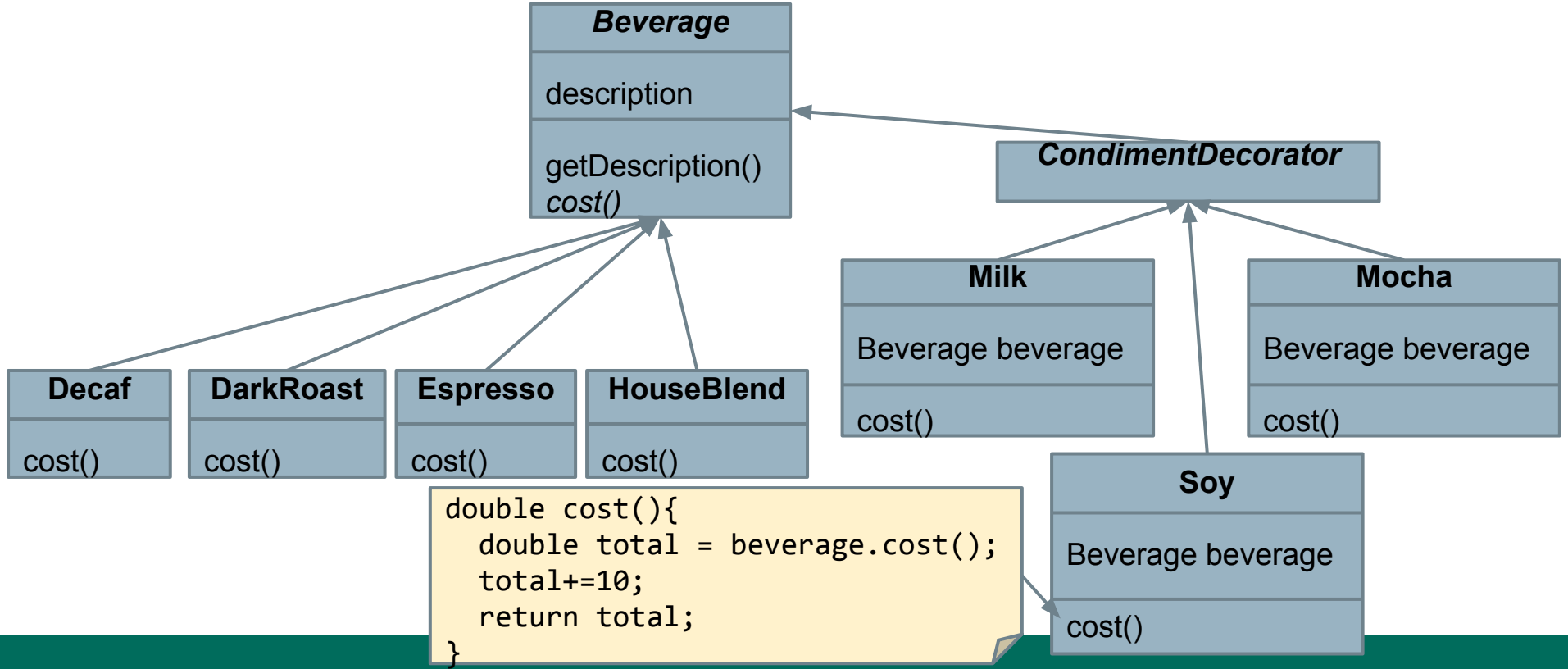


Each Decorator offers same methods the Component offers.

Each concrete Decorator has instance variable to store wrapped component.

Decorators add behavior by adding operations and attributes.

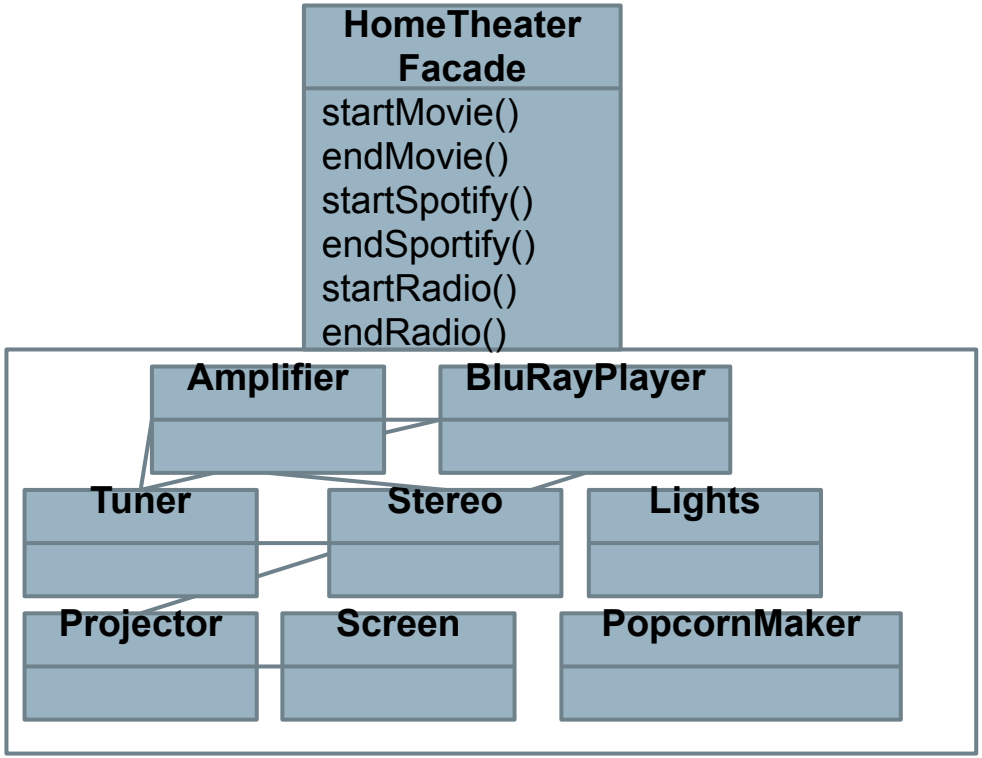
Ordering System - Decorator Pattern



The Open-Closed Principle

- **Classes should be open for extension, closed for modification.**
  - Add new behavior without changing existing code.
  - Create class with new data and operators, attach class it is intended to extend.
  - Allow extension without direct modification.
- **Do not try to apply this everywhere**.
  - Focus on areas likely to change.

The Facade Pattern



- Create a new class that exposes simple methods (the **facade**).
- Facade calls on classes to implement high-level methods.
- Client calls facade instead of classes.
- Classes still accessible.

The Principle of Least Knowledge

- **Talk only to your immediate friends**.
- Be careful of the number of classes your class interacts with and how it interacts with them.
- Only invoke methods that belong to the object, objects passed as parameters, objects created or instantiated, and attached objects.

Let’s Take a Break

Modular implementation: components and services

Frameworks

- A collection of classes that represent solutions to related problems.
  - Base implementation that can be extended with *plug-ins*, supporting new custom use cases.
  - Provides extension points (“**hot spots**”)
- Framework is responsible for main control flow, asks plug-ins for custom behavior.

Variability with frameworks

- Composition-based, load-time.
- White Box: Subclass an abstract parent, override template methods to implement features.
- Black Box: Register plug-in objects that provide specific features.
- Provides clear modularity, but requires extensive up-front design effort.

Components

- A **component** is a standalone unit with specified interfaces and explicit dependencies.
  - Can be deployed independently.
  - Can be reused in many systems.
  - Can vary from one class to many.
- Developers can choose to implement their own components or work with existing ones.
  - Requires compatible interfaces and data.

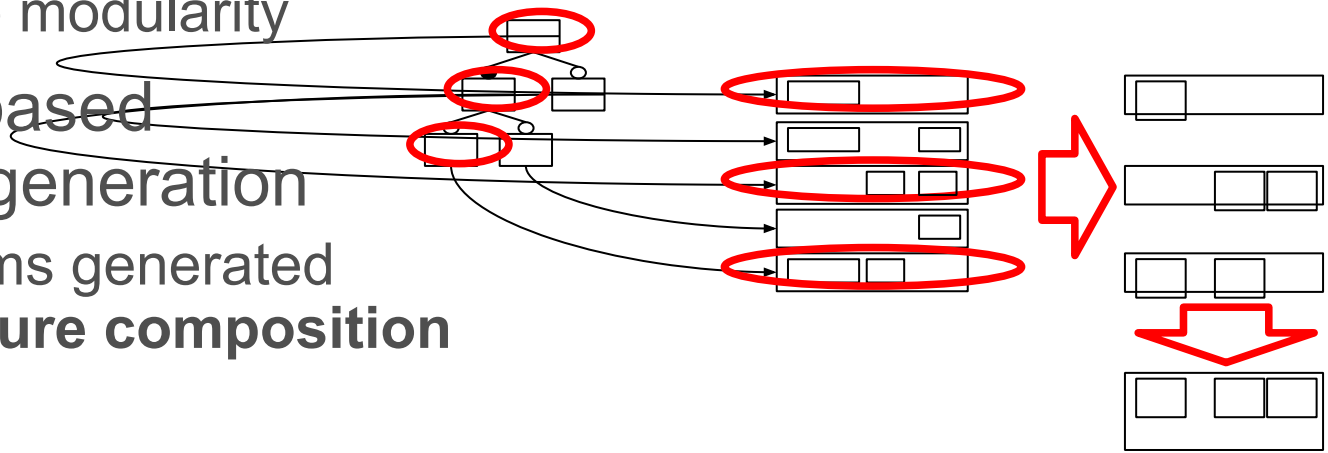
Variability with components

- For features that offer standalone functionality with explicit interface and dependencies.
- Interfaces often standardized (REST), leading to services.
- Can be reused in many projects.
- Integrated as part of a broader architectural design.

Feature- and Aspect-Oriented Programming

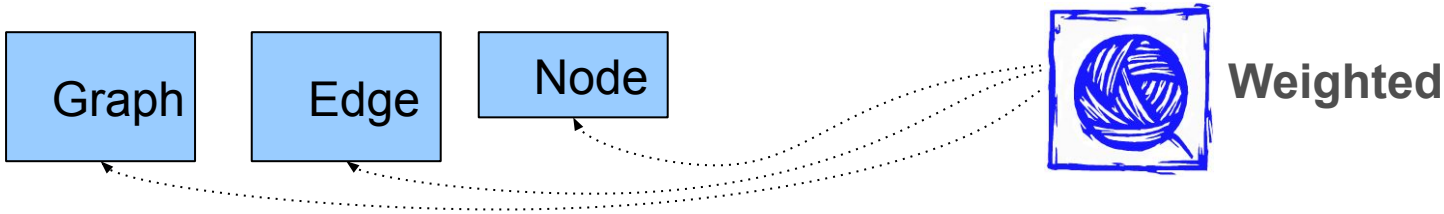
Feature orientation

- Language-based approach for feature traceability
- Implement each feature in a feature module
  - Perfect feature traceability
  - Feature modularity
- Feature-based program generation
  - Programs generated via **feature composition**

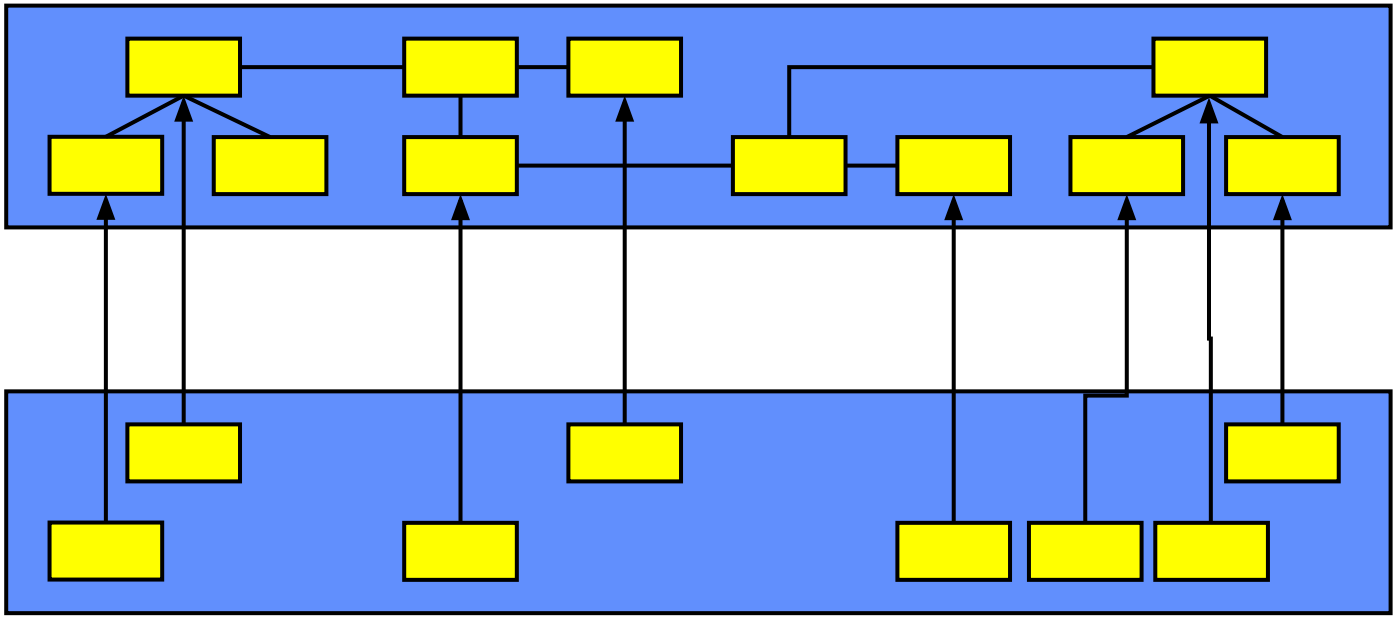


Aspect orientation

- Modularize a cross-cutting concern into an aspect
- Aspect describes effects on rest of software
- How interpreted? Multiple options:
  - as a program transformation
  - as a metaobject protocol
  - some sort of feature module

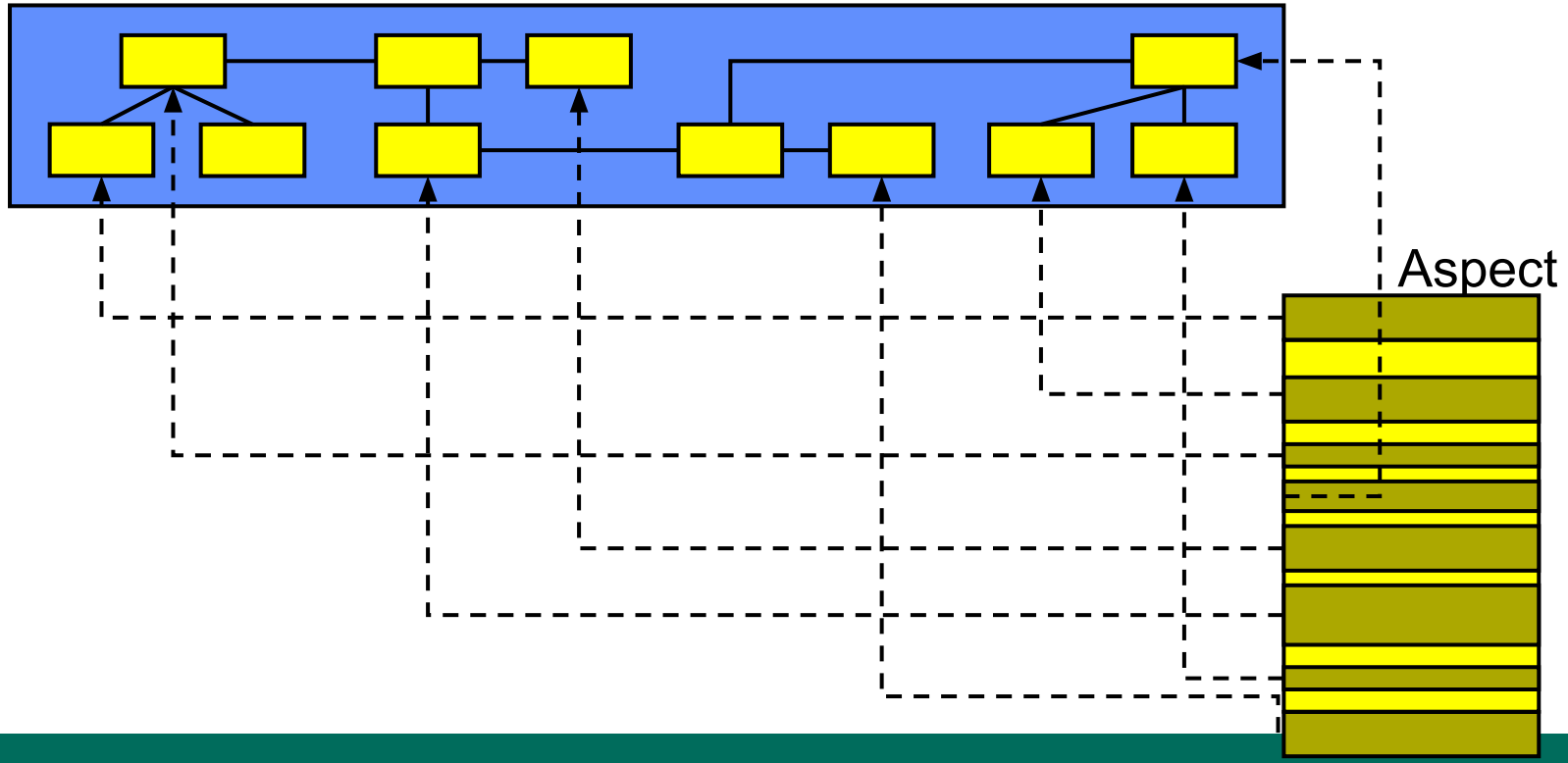


AOP vs. FOP



Collaboration

AOP vs. FOP



	FOP	AOP
static	*good support* – fields, method, classes	*limited support* – fields, methods, static inner classes
dynamic	*bad support* – only simple extensions (method refinement)	*good support* – advanced extensions, thanks to language support for dealing with execution context
hetero-geneous	*good support* – refinements and collaborations	*limited support* – possible, but object-oriented structure gets lost and aspects can get huge
homo-geneous	*no support* – one refinement per join point (might lead to code replication)	*good support* – wildcards and logical quantification over pointcuts

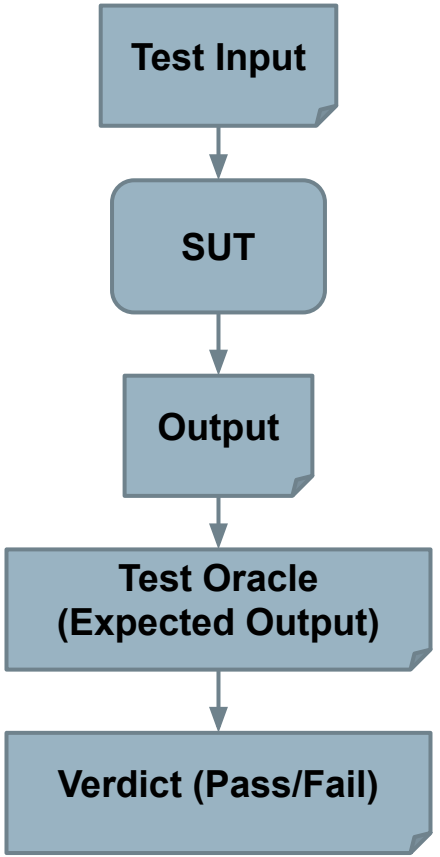
Wrap up: implementation techniques

Preprocessors	Compile-Time	Tool-Based	Annotation-Based
Build Systems	Compile-Time	Tool-Based	Composition-Based
Parameters	Load or Run-Time	Language-Based	Annotation-Based
Design Patterns	Load or Run-Time	Language-Based	Composition-Based
Frameworks	Load or Run-Time	Language-Based	Composition-Based
Components	Any	Any	Composition-Based
FOP	Compile-Time	Language-Based	Composition-Based
AOP	Any	Language-Based	Composition-Based

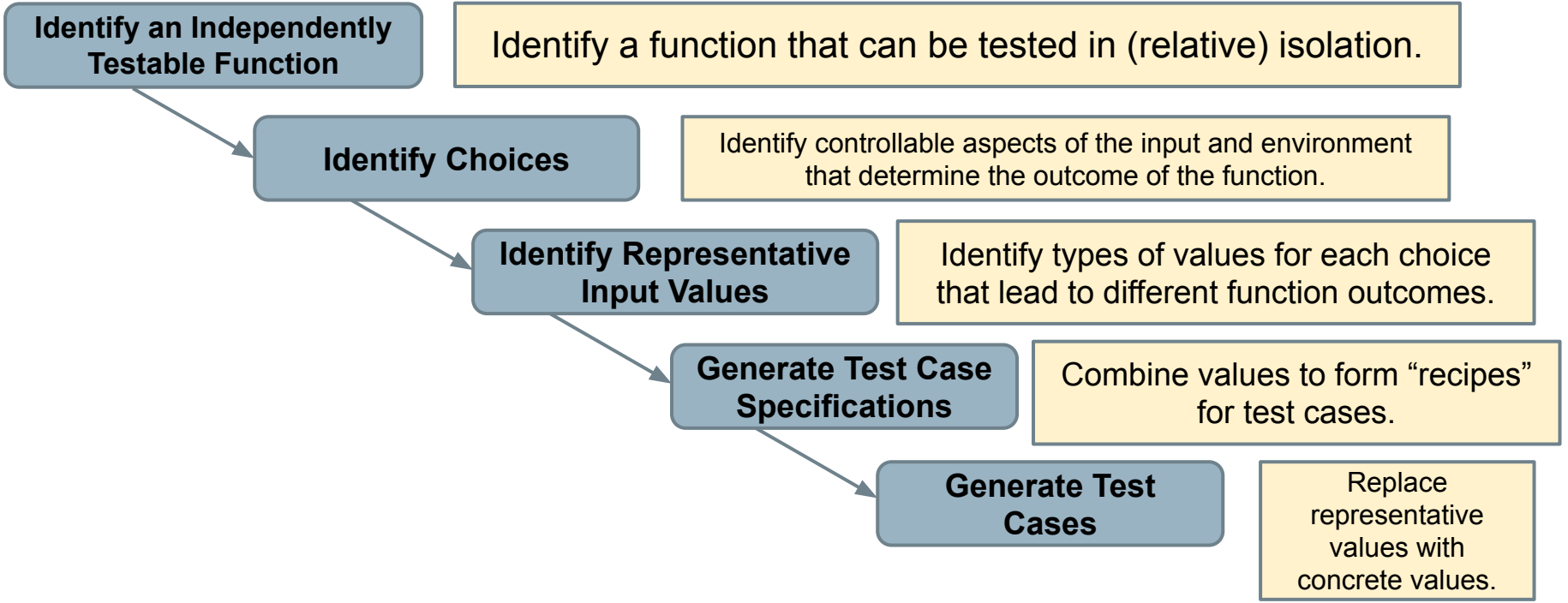
Designing Test Cases

Software Testing

- An investigation into system quality.
- Based on sequences of **stimuli** and **observations**.
  - **Stimuli** that the system must react to.
  - **Observations** of system reactions.
  - **Verdicts** on correctness.



Creating System-Level Tests



Example - Register for a Class

Test Choices

- **Parameter: studentID**
  - Validity of Student ID
  - Courses Student Has Taken Previously
- **Parameter: courseID**
  - Validity of Course ID
  - Prerequisites of Course ID

Choosing Input Partitions

- Equivalent output events.
- Ranges of numbers or values.
- Membership in a logical group.
- Time-dependent equivalence classes.
- Equivalent operating environments.
- Data structures.
- Partition boundary conditions.

Example - Register for a Class

Parameter: studentID

- Validity of Student ID
  - Active Student
  - Inactive Student
  - Non-Existent Student
- Courses Student Has Taken Previously
  - Matches Prerequisites
  - Does Not Match Prerequisites

Parameter: courseID

- Validity of Course ID
  - Existing Course
  - Non-Existent Course
- Prerequisites of Course ID
  - Only Courses Taken By Student
  - Only Courses Not Taken By Student
  - Some Courses Taken by Student

Forming Specification

Parameter: studentID

- Validity of Student ID
  - Active Student
  - Inactive Student
  - Non-Existent Student
- Courses Student Has Taken Previously
  - Matches Prerequisites
  - Does Not Match Prerequisites

Parameter: courseID

- Validity of Course ID
  - Existing Course
  - Non-Existent Course
- Prerequisites of Course ID
  - Only Courses Taken By Student
  - Only Courses Not Taken By Student
  - Some Courses Taken by Student

Test Specifications:

- Active, Matches, Existing, Only Taken
- Active, Does Not Match, Existing, Only Not Taken
- Active, Does Not Match, Existing, Some Taken
- Active, - , Non-Existing, -
- Inactive, Matches, Existing, Only Taken
- Inactive, Does Not Match, Existing, Only Not Taken
- Inactive, Does Not Match, Existing Some Taken
- Inactive, - , Non-Existing, -
- Non-Existing, -, Existing, -
- Non-Existing, -, Non-Existing, -
- …

Specifications: 3 * 2 * 2 * 3 = 36 - Illegal Combinations

Generate Test Cases

```
@Test
public void testRegistration() {
    // Set Up
    setupStudentRecord(ggay, active, [TDA050, TDA360]);
    setupCourse(TDA594, [TDA360]),
    // Attempt to register for a course
    Boolean outcome = registerForCourse(ggay, TDA594);
    Boolean expected = true;
    // Check the result of registration
    assertEquals(expected, outcome);
}
```

Specification:

Active, Matches, Existing, Only Taken

- Fill in concrete values that match the representative values classes.
- Can create MANY concrete tests for each specification.

Example - Set Functions

Generate Test Case Specifications

Set Size	Obj in Set	Obj Status	Outcome
Empty	Yes	Valid	No change
Empty	Yes	Null	Error
Empty	No	Valid	Obj added to Set
Empty	No	Null	Error
1 item	Yes	Valid	No change
1 item	Yes	Null	Error
1 item	No	Valid	Obj added to Set
1 item	No	Null	Error
10 items	Yes	Valid	No change
10 items	Yes	Null	Error
10 items	No	Valid	Obj added to Set
10 items	No	Null	Error

```
void insert(Set set,
Object obj)
```

- (4 * 2 * 2) = 16 specifications
- Each can become 1+ tests.
- Use constraints to remove impossible combinations.

Set Size	Obj in Set	Obj Status	Outcome
10000	Yes	Valid	No change (may be slowdown)
10000	Yes	Null	Error
10000	No	Valid	Obj added to Set(may be slowdown)
10000	No	Null	Error (may be slowdown)

Constraints Between Values

- IF-CONSTRAINT
  - This value only needs to be used under certain conditions (**if X is true, use value Y**)
- ERROR
  - Value causes error regardless of values of other choices.
- SINGLE
  - Only a single test with this value is needed.
  - Corner cases that should give “good” outcome.

Example - Set Functions

```
void insert(Set set, Object obj)
```

Identify Constraints

Parameter: set

- **Choice:** How many items are in the set?
  - **Representative Values:**
    - Empty Set — property empty
    - Set with 1 item
    - Set with 10 items — single
    - Set with 10000 items — single

Parameter: obj

- **Choice:** Is the object already in the set?
  - **Representative Values:**
    - obj already in set — if !empty
    - obj not in set
- **Choice:** Is the object valid?
  - **Representative Values:**
    - Valid obj
    - Null obj — error

Example - Set Functions

Apply Constraints

Set Size	Obj in Set	Obj Status	Outcome
~~Empty~~	~~Yes~~	~~Valid~~	~~No change~~
~~Empty~~	~~Yes~~	~~Null~~	~~Error~~
Empty	No	Valid	Obj added to Set
Empty	No	Null	Error
1 item	Yes	Valid	No change
~~1 item~~	~~Yes~~	~~Null~~	~~Error~~
1 item	No	Valid	Obj added to Set
~~1 item~~	~~No~~	~~Null~~	~~Error~~
~~10 items~~	~~Yes~~	~~Valid~~	~~No change~~
~~10 items~~	~~Yes~~	~~Null~~	~~Error~~
10 items	No	Valid	Obj added to Set
~~10 items~~	~~No~~	~~Null~~	~~Error~~

```
void insert(Set set,
Object obj)
```

(4 * 2 * 2) = 16 specifications

Can't already be in empty set, - 2

error (null), - 6

single (10, 10000), - 2

Set Size	Obj in Set	Obj Status	Outcome
~~10000~~	~~Yes~~	~~Valid~~	~~No change (may be slowdown)~~
~~10000~~	~~Yes~~	~~Null~~	~~Error (may be slowdown)~~
10000	No	Valid	Obj added to Set(may be slowdown)
~~10000~~	~~No~~	~~Null~~	~~Error (may be slowdown)~~

Example - Set Functions

Apply Constraints

Set Size	Obj in Set	Obj Status	Outcome
Empty	No	Valid	Obj added to Set
Empty	No	Null	Error
1 item	Yes	Valid	No change
1 item	No	Valid	Obj added to Set
10 items	No	Valid	Obj added to Set
10000	No	Valid	Obj added to Set(may be slowdown)

```
void insert(Set set,
Object obj)
```

- From 16 -> 6 specifications
- Each can become 1+ tests.
- Can further constrain if needed.

Example - Set Functions

`void insert(Set set, Object obj)`

Create Test Cases

Set Size	Obj in Set	Obj Status	Outcome
Empty	No	Valid	Obj added to Set

Set Size	Obj in Set	Obj Status	Outcome
Empty	No	Null	Error

```
@Test
public void testInsertEmptyValid() {
    // Set up the existing set
    Set target = new Set();
    // Insert an object
    String obj = “Test”;
    insert(target, obj);
    // Check the result
    assertTrue(find(target,obj));
}
```

```
@Test
public void testInsertemptyNull() {
    // Set up the existing set
    Set target = new Set();
    // Insert null object and check exception
    Throwable exc = assertThrows(
        SetException.class, () -> {
            insert(target, null); });
    assertEquals("Null Object", exc.getMessage());
}
```

Wrap-Up

- ***Thank you for making this a great course!***
- Any remaining questions?

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY