Lecture 24

ECE 1145: Software Construction and Evolution

Final Exam Review

Course Wrap-Up

Announcements

- Iteration 8 (last one!): Frameworks and MiniDraw due Dec. 12
- OMET Teaching Surveys open until Dec. 12
  - Link in Canvas navigation
- Final Exam: 12:00 AM Wed. Dec. 15 – 11:59 PM Fri Dec. 17 (similar format as midterm)
  - Office hours during scheduled exam time 10:00 – 11:50 AM Wed. in 1211A

Final Exam Topics

- Focus on Lectures 10 – 23
- Variability Management, State Pattern (Lecture 10)
- Test Stubs (Lecture 11)
- Abstract Factory, Pattern Fragility (Lecture 12)
- Compositional Design, Roles and Responsibility (Lecture 14)
- Multi-Dimensional Variance (Lecture 15)
- Design Pattern Catalog (Lecture 16)
  - Façade, Decorator, Adapter, Builder, Command, Proxy, Null Object
- Systematic Testing (Lecture 17)
- Code Coverage (Lecture 18)
- More Patterns (Lecture 19)
  - Composite, Observer, Model-View Controller, Template Method
- Framework Theory (Lecture 20)
- MiniDraw, HotCiv GUI (Lecture 21, 22)
- Error Handling (Lecture 23)

GammaTown: Alternating Rate

GammaTown wants “almost the same” pay station, but with different rate calculations during weekdays and weekends:

- Weekdays: Linear rate (like AlphaTown)
- Weekends: Progressive rate (like BetaTown)

Approaches:

1. Source tree copy
2. Parametric: Add Gammatown case to all switch statements
3. Polymorphic: Subclass PayStation
4. Compositional + parametric: If statement in PayStation selects the strategy
5. Compositional: Object collaboration, delegation

GammaTown: Compositional Proposal

Recall 3-1-2 process:

(3): Identify behavior that varies

→ Rate calculation

(1): State a responsibility that covers the behavior and express it as an interface

→ RateStrategy (already exists)

(2): Compose the behavior by delegating

→ Implement GammaTown behavior by combining existing rate calculations

→ Proposal: A “coordinator” delegates rate calculation to specialized “workers” (already-implemented rate calculations)

GammaTown: Compositional Proposal

Key Point: Object collaborations define compositional designs

When designing software compositionally, you make objects collaborate to achieve complex behavior.

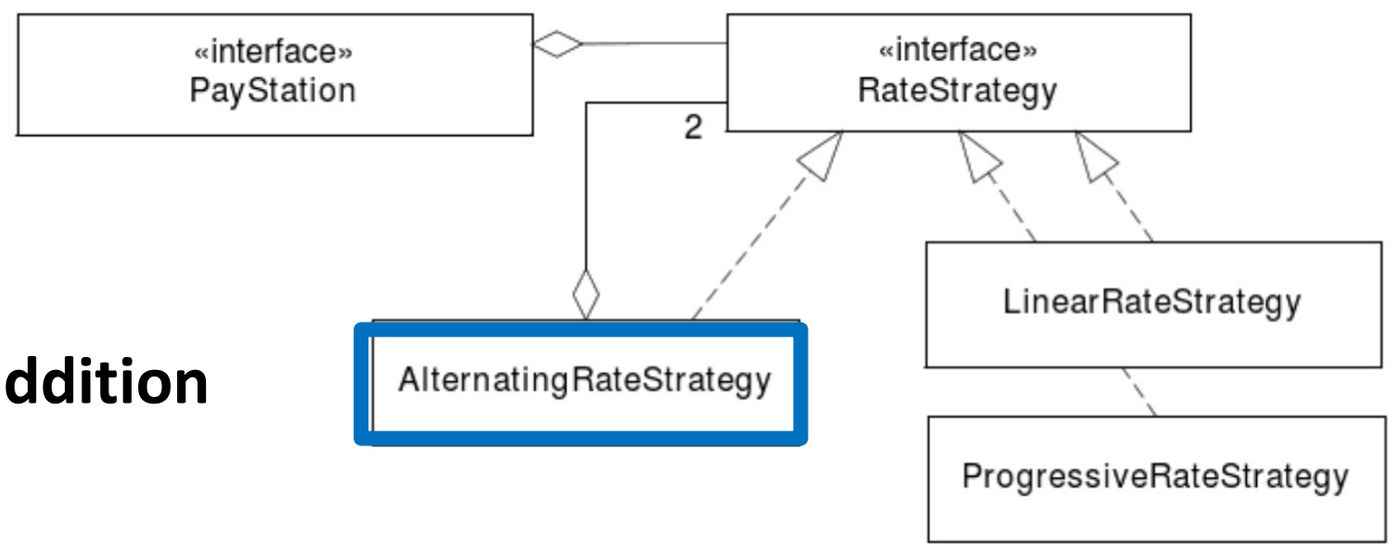


Figure 11.3: Rate calculation as a combined effort.

Change by **addition**

GammaTown: Compositional Design

```
package paystation.domain;

import java.util.*;

/** A rate strategy that uses the State pattern to vary behavior
    according to the state of the system clock: a linear rate
    during weekdays and a progressive rate during weekdends.
*/
public class AlternatingRateStrategy implements RateStrategy {
  private RateStrategy
    weekendStrategy, weekdayStrategy, currentState;
  public AlternatingRateStrategy( RateStrategy weekdayStrategy,
                                  RateStrategy weekendStrategy ) {
    this.weekdayStrategy = weekdayStrategy;
    this.weekendStrategy = weekendStrategy;
    this.currentState = null;
  }
  public int calculateTime( int amount ) {
    if ( isWeekend() ) {
      currentState = weekendStrategy;
    } else {
      currentState = weekdayStrategy;
    }
    return currentState.calculateTime( amount );
  }
```

This is the **State** design pattern

Decide **state** of RateStrategy

```
  private boolean isWeekend() {
    Date d = new Date();
    Calendar c = new GregorianCalendar();
    c.setTime(d);
    int dayOfWeek = c.get(Calendar.DAY_OF_WEEK);
    return ( dayOfWeek == Calendar.SATURDAY
             ||
             dayOfWeek == Calendar.SUNDAY);
  }
}
```

Delegate calculation

Compositional Design: Analysis

- **Reliability**: this will not affect AlphaTown and BetaTown implementations
  - Return later to automated testing
- **Maintainability**: simple and straightforward implementation
- **Client's interface is consistent**: pay station use of rate calculations has not changed
- **Reuse**: can be applied to any future customers wanting weekday/weekend variation
- **Flexibility**: can change by addition for new requirements
- **Cohesion**: state-specific behavior is localized to AlternatingRateStrategy

- Increased number of objects

```
package paystation.domain;

import java.util.*;

/** A rate strategy that uses the State pattern to vary behavior
    according to the state of the system clock: a linear rate
    during weekdays and a progressive rate during weekdends.
*/
public class AlternatingRateStrategy implements RateStrategy {
  private RateStrategy
    weekendStrategy, weekdayStrategy, currentState;
  public AlternatingRateStrategy( RateStrategy weekdayStrategy,
                                  RateStrategy weekendStrategy ) {
    this.weekdayStrategy = weekdayStrategy;
    this.weekendStrategy = weekendStrategy;
    this.currentState = null;
  }
  public int calculateTime( int amount ) {
    if ( isWeekend() ) {
      currentState = weekendStrategy;
    } else {
      currentState = weekdayStrategy;
    }
    return currentState.calculateTime( amount );
  }
```

```
  private boolean isWeekend() {
    Date d = new Date();
    Calendar c = new GregorianCalendar();
    c.setTime(d);
    int dayOfWeek = c.get(Calendar.DAY_OF_WEEK);
    return ( dayOfWeek == Calendar.SATURDAY
             ||
             dayOfWeek == Calendar.SUNDAY);
  }
}
```

```
public void setUp() {
  RateStrategy rs =
    new AlternatingRateStrategy( new LinearRateStrategy(),
                                 new ProgressiveRateStrategy() );

  ps = new PayStationImpl( rs );
}
```

State Pattern

State pattern: allow an object to alter its behavior when its internal state changes

- **Context** object delegates requests to the state object
  - AlternatingRateStrategy
- Internal state changes change the concrete **state object**
  - LinearRateStrategy
  - ProgressiveRateStrategy

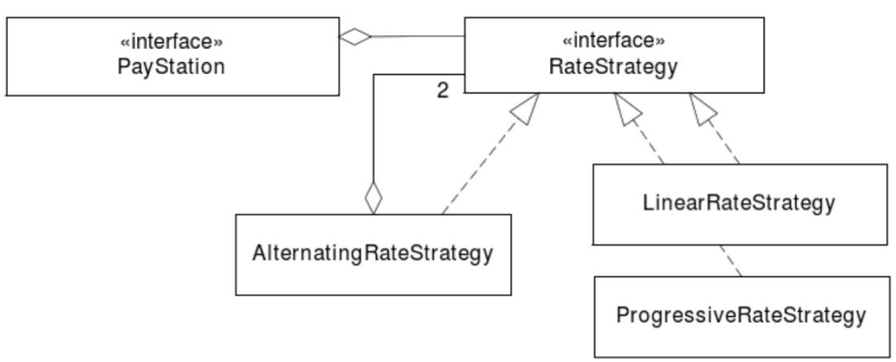


Figure 11.3: Rate calculation as a combined effort.

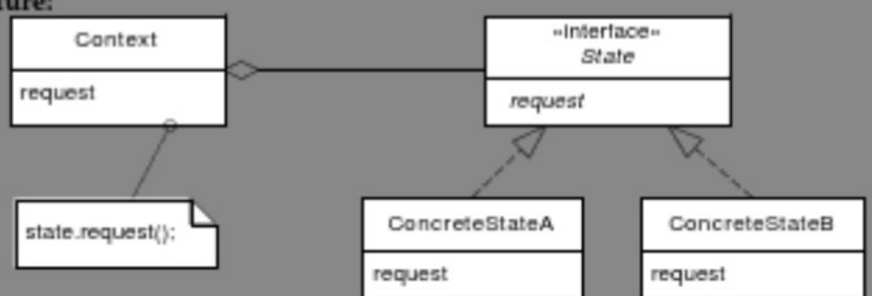
[11.1] Design Pattern: State

Intent Allow an object to alter its behavior when its internal state changes.

Problem Your product's behavior varies at run-time depending upon some internal state.

Solution Describe the responsibilities of the dynamically varying behavior in an interface and implement the concrete behavior associated with each unique state in an object, the state object, that implements this interface. The context object delegates to its current state object. When internal state changes occur, the current state object reference is changed to refer to the corresponding state object.

Structure:



Roles **State** specifies the responsibilities and interface of the varying behavior associated with a state, and **ConcreteState** objects define the specific behavior associated with each specific state. The **Context** object delegates to its current state object. The state object reference is changed whenever the context changes its internal state.

Cost - Benefit *State specific behavior is localized* as all behavior associated with a specific state is in a single class. It *makes state transitions explicit* as assigning the current state object is the only way to change state. A liability is the *increased number of objects and interactions* compared to a state machine based upon conditional statements in the context object.

Direct/Indirect Inputs

Direct input is values or data that affects the behavior of the unit under test that can be provided **directly** by the testing code

Indirect input is values or data that affects the behavior of the unit under test that **cannot** be provided directly by the testing code

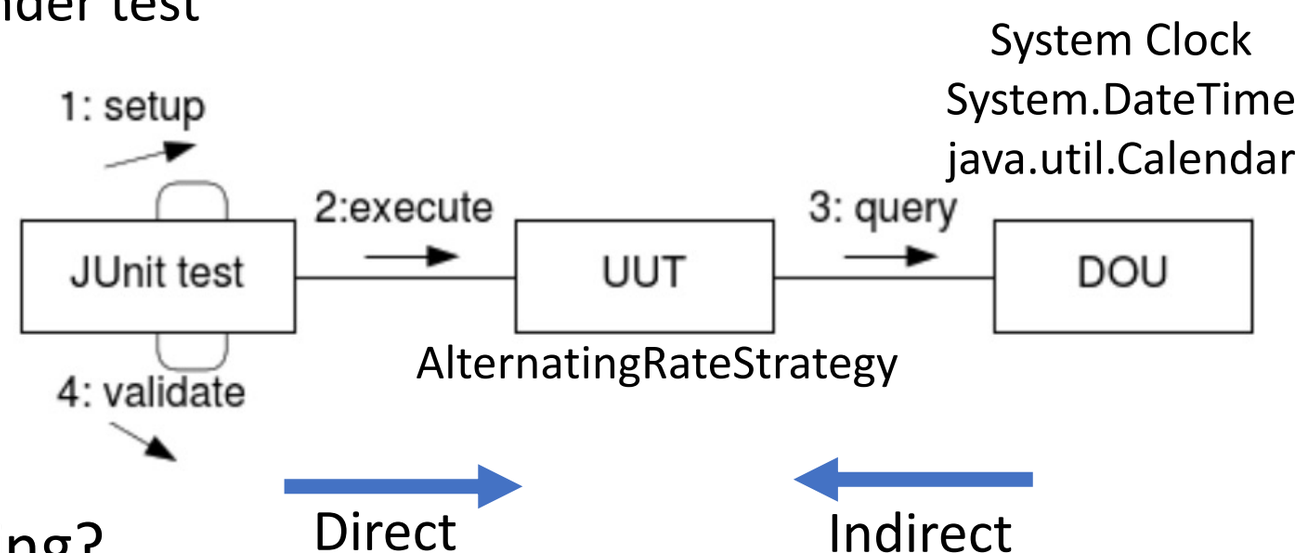
A **depended-on unit** is a unit in the production code that provides values or behavior that affect the unit under test

UUT: Unit under test

DOU: Depended-on unit

Day of the week is an **indirect input parameter.**

How do we automate testing?



Test Stub: Indirect Input Testing

A **test stub** is a replacement of a real depended-on unit that feeds indirect input (defined by the test code) into the unit under test

Create WeekendDecisionStrategy to separate this responsibility from AlternatingRateStrategy

→ Why isn't this State? We are **choosing an algorithm** to check if it is the weekend

→ Enables automated unit testing of AlternatingRateStrategy using a **Test Stub**

- FixedDecisionStrategy

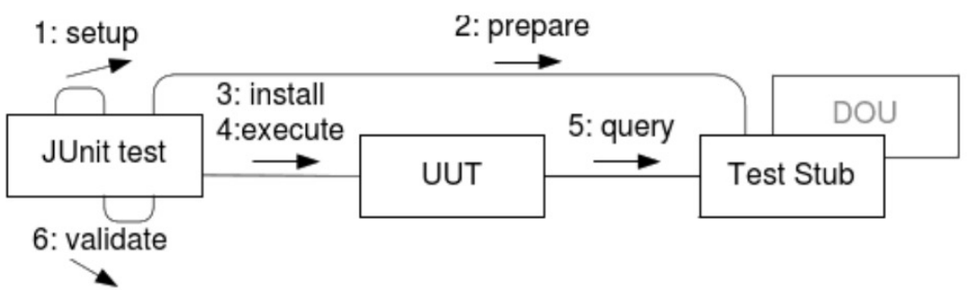
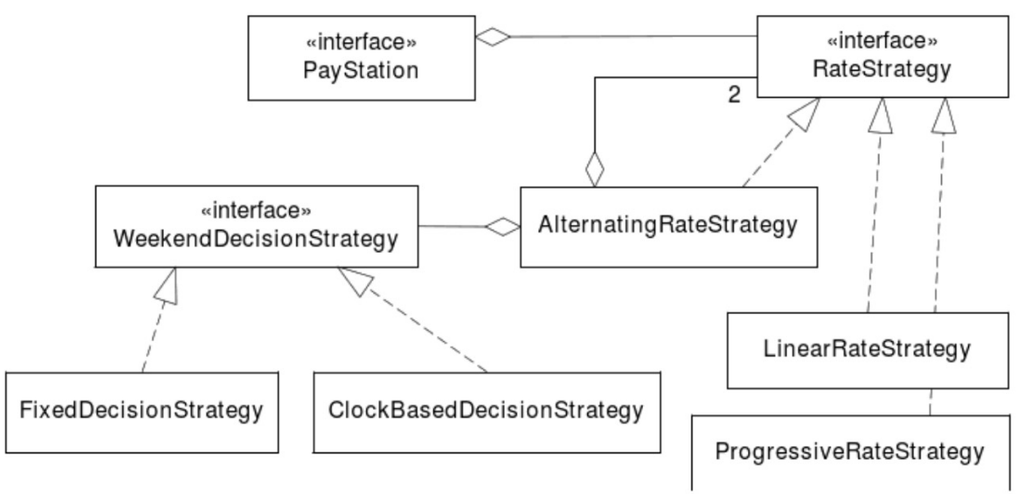


Figure 12.3: Test Stub replacing the DOU.



Dependency Injection

This technique is enabled by **compositional design** and proper **encapsulation** of behavior that provides the indirect input

- Only possible if depended-on unit is not created by or otherwise tightly coupled to the UUT
- “Inject your dependencies” - pass values to the UUT rather than having the UUT create them, to enable testing

Becomes increasingly important with greater complexity

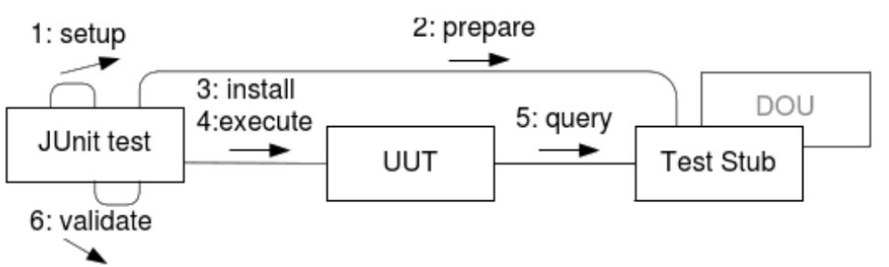


Figure 12.3: Test Stub replacing the DOU.

Test Doubles

A Test Stub is a subtype of **Test Double**

Types of Test Doubles:

- **Test Stub:** a double that feeds **indirect input** (defined by the test case) into the UUT
- **Test Spy:** a double that records the UUT's indirect output for later verification by the test case
- **Fake:** a double that acts as a high-performance replacement for a slow or expensive DOU
- **Mock object:** a double created and programmed dynamically by a mock library that may serve as both a stub and a spy

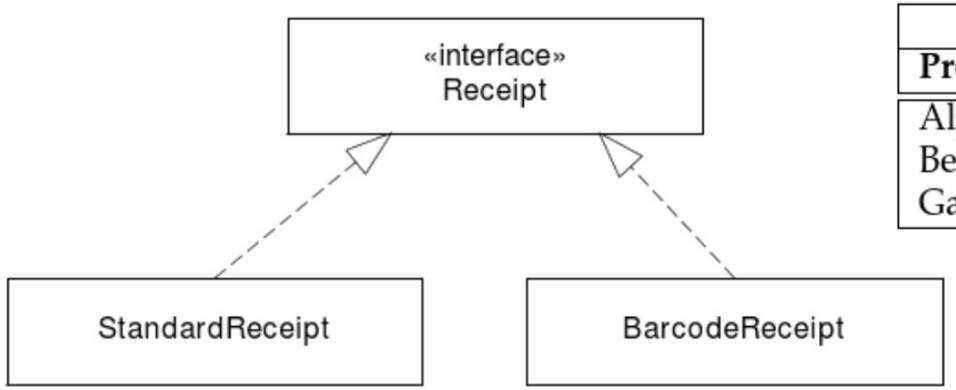
Test doubles make software **testable** by replacing real units and allowing test code to control **indirect input**, detect **indirect output**, or act as a **mimic** of a slow/expensive external resource

New Customer Request!

BetaTown wants printed receipts with parking statistics in the form of **bar codes**

```
-----------------------------------------------
------- P A R K I N G   R E C E I P T  -------
              Value 049 minutes.
            Car parked at 08:06
-----------------------------------------------
```

```
-----------------------------------------------
------- P A R K I N G   R E C E I P T  -------
              Value 049 minutes.
            Car parked at 08:06
||  ||||| | || ||| || ||  ||| | || |||| | || ||||
-----------------------------------------------
```



	Variability points	
Product	**Rate**	**Receipt**
Alphatown	Linear	Standard
Betatown	Progressive	Barcode
Gammatown	Alternating	Standard

Figure 13.1: New types of Receipts.

First Attempt: 3-1-2

(3) Identify a behavior that needs to vary

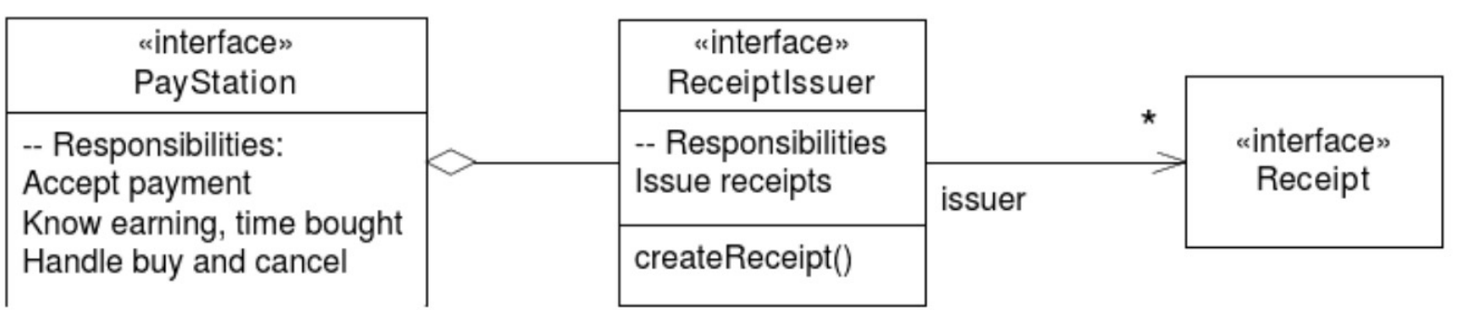
→ Printing of receipts; printing is instantiation, so instantiation needs to vary. AlphaTown and GammaTown Pay Stations should instantiate StandardReceipt, while BetaTown Pay Stations should instantiate BarcodeReceipt

(1) State a responsibility that covers the behavior and express it as an interface

→ We can define a ReceiptIssuer interface whose responsibility is to issue Receipts (that is, create Receipt objects). Currently this is handled by PayStation

(2) Compose the resulting behavior by delegating

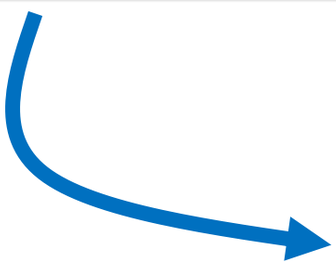
→ The pay station should delegate to the receipt issuer to instantiate receipts



First Attempt: 3-1-2

Current test fixture:

```
PayStation ps;
/** Fixture for pay station testing. */
@Before
public void setUp() {
  ps = new PayStationImpl( new One2OneRateStrategy() );
}
```

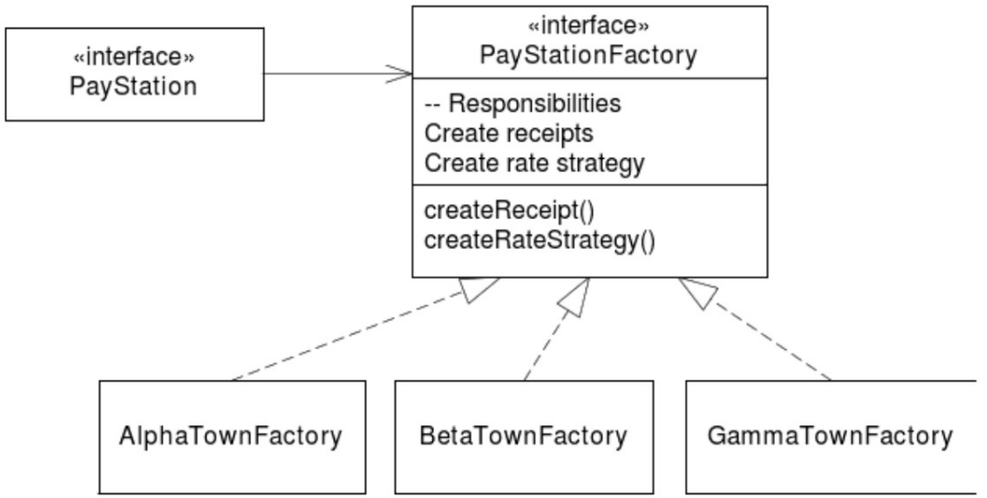


```
PayStation ps;
/** Fixture for pay station testing. */
@Before
public void setUp() {
  ps = new PayStationImpl( new One2OneRateStrategy(),
                           new StandardReceiptIssuer() );
}
```

Problem: Configuration behavior is split between two different objects, the pay station and receipt issuer

→ Not cohesive

New Attempt



Create a **factory object**

→ Responsibility is to make objects

```
public class PayStationImpl implements PayStation {
  [...]
  /** the strategy for rate calculations */
  private RateStrategy rateStrategy;
  /** the factory that defines strategies */
  private PayStationFactory factory;

  /** Construct a pay station.
      @param factory the factory to produce strategies and receipts
  */
  public PayStationImpl( PayStationFactory factory ) {
    this.factory = factory;
    this.rateStrategy = factory.createRateStrategy();
    reset();
  }
  [...]
  public Receipt buy() {
    Receipt r = factory.createReceipt(timeBought);
    reset();
    return r;
  }
  [...]
}
```

Refactor PayStationImpl to use the factory

Dependency injection! Enables unit testing of PayStation with TestTownFactory

```
  public RateStrategy createRateStrategy() {
    return new One2OneRateStrategy();
  }
  public Receipt createReceipt( int parkingTime ) {
    return new StandardReceipt(parkingTime);
  }
}
```

Abstract Factory

Abstract Factory is a solution to the problem of creating variable types of objects

Pros:

- Low coupling between client and products
  - Only communicate through **interfaces**
- Configuring clients is easy
  - Provide client with the proper **concrete factory**
- Promotes consistency among products
  - Factories **encapsulate configuration**, easier to avoid or track defects
- Change by addition, not modification
  - Easy to introduce **new pay station types** (new concrete factories)
- Client constructor parameter list stays intact
  - Client's constructor **only takes the factory object** as its parameter

Cons:

- Introduces extra classes and objects
  - Complex, unnecessary for a single variant
- Introducing **new aspects of variation** is problematic
  - Need to **modify** the abstract factory interface and all concrete factory subclasses

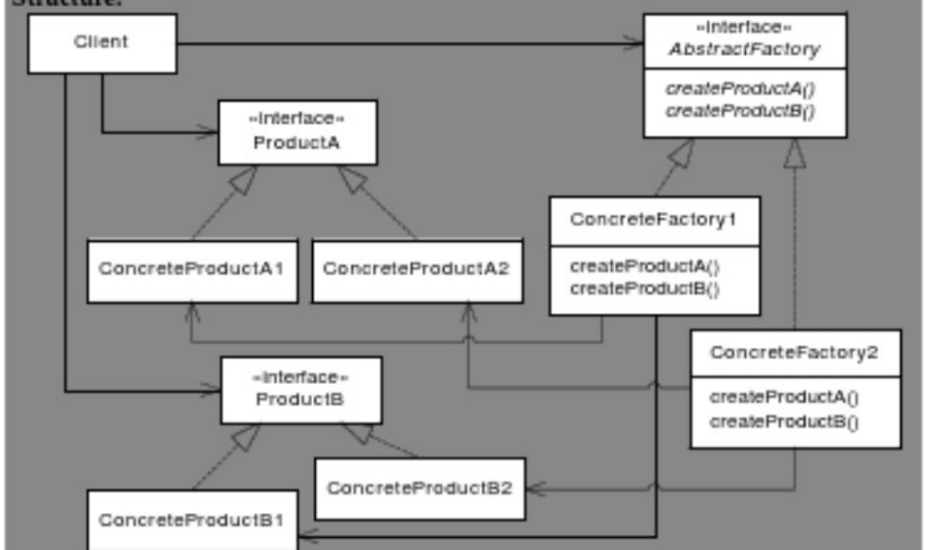
[13.1] Design Pattern: Abstract Factory

Intent Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Problem Families of related objects need to be instantiated. Product variants need to be consistently configured.

Solution Define an abstraction whose responsibility it is to create families of objects. The client delegates object creation to instances of this abstraction.

Structure:



Roles **Abstract Factory** defines a common interface for object creation. **ProductA** defines the interface of an object, **ConcreteProductA1**, (product A in variant 1) required by the client. **ConcreteFactory1** is responsible for creating **Product**s that belong to the variant 1 family of objects that are consistent with each other.

Cost - Benefit *It lowers coupling between client and products* as there are no new statements in the client to create high coupling. *It makes exchanging product families easy* by providing the client with different factories. *It promotes consistency among products* as all instantiation code is within the same class definition that is easy to overview. However, *supporting new kinds of products is difficult*: every new product introduced requires all factories to be changed.

Pattern Fragility

Recall: The advantage of patterns is that they have been proven through experience to solve certain problems, with known benefits and liabilities

But, for patterns to be effective, It is important to have the correct implementation of the pattern **structure** and **interactions**!

→ Coding practices or mistakes can **invalidate the benefits** of patterns

→ Know the patterns, know the **roles** and **interactions** they involve

→ Know the implications of **how they are coded**

→ Understand the patterns used in a given piece of code **before making changes**

→ Make sure everyone working on a project understands the design!

Objects: Responsibility-Centric Perspective

An object is something with **responsibilities**.

- Focus is on a program's **dynamics** (each object's behavior and responsibilities)
- Relates well to requirements/specifications: **Functionality** is most important from a customer perspective, not how closely our program resembles the real world

An program is structured as a community of **interacting objects**. Each object has a **role** to play; each object provides a **service** or performs an action that is used by other members of the **community**.

Addition of RateStrategy to pay station was responsibility-centric!

Roles, Responsibility, Behavior

Behavior: acting a particular and observable way ("doing something")

- Methods are **templates** for behavior
- Collective behavior arises when objects **interact**, request behavior from other objects

Responsibility: being accountable and dependable to answer a request.

- Related to behavior, but more abstract
- Well described by **interfaces**

Protocol: a convention detailing the expected sequence of interactions or actions by a set of roles

Role: a set of responsibilities and associated protocols

Class name **C**ollaborators

PayStation	
Accept payment Calculate parking time based on payment Know earning, parking time bought Issue receipts Handle buy and cancel events	PayStationHardware Receipt

Responsibilities

CRC Card

Example: Roles in HotCiv

Game (coordinator/manager)

- Role: Is responsible for overall game mechanics, collaborates with lots of other roles

(World)

- Role: Primary state holder of game world + simple state changes

Unit, City (specialists)

- Role: Primary state holders + simple, local, state changes (owner)

WinnerStrategy (super-specialist)

- Role: Is responsible for calculating who has won
  - Access information from other roles (e.g., Game, Cities) to do the calculation

WorldLayoutStrategy

- Role: Is responsible for creating a world

Responsibility-Centric Design

Model-centric: focus on structure first, then behavior

- "Who/what" cycle

Responsibility-centric: define functionality in terms of responsibilities and roles, then assign objects to roles

- "What/who" cycle
- Group tasks into cohesive roles that collaborate with sensible protocols
- Design software as a **community** of interacting objects (roles)

Compositional Design

Recall 3-1-2 process:

③ I identified some behavior that was likely to change…

① I stated a well defined responsibility that covers this behavior and expressed it in an interface…

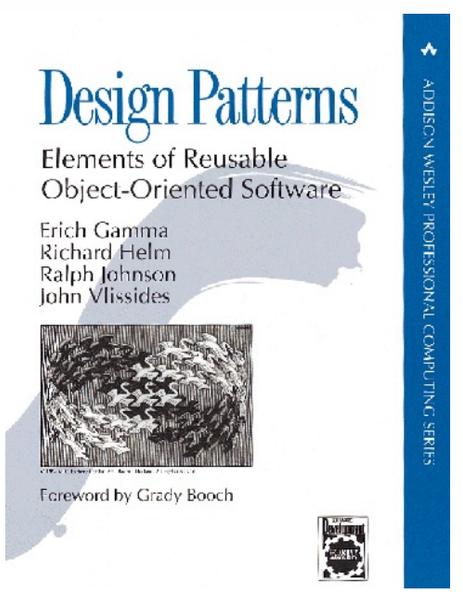
② Instead of implementing the behavior ourselves I delegated to an object implementing the interface…

Principles for Flexible Design:

① *Program to an interface, not an implementation.*

② *Favor object composition over class inheritance.*

③ *Consider what should be variable in your design.*
(or: Encapsulate the behavior that varies.)



Gamma et al. 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*

Multi-Dimensional Variation

AlphaTown wants the pay station to display the **time when parking expires** (instead of the number of minutes, as previously)

How does this add to our current dimensions of variation?

→ **Rate calculation**: linear, progressive, alternating

→ **Receipt format:** standard, barcode

→ **Display output**: minutes of parking time, time when parking ends (new!)

3 Independent Dimensions!

→ **All combinations are valid**

- 3 rate policies x 2 receipt types x 2 display options = 12 variants
- More policies could lead to combinatorial explosion

Configuration Table

	Variability points		
Product	**Rate**	**Receipt**	**Display**
Alphatown	Linear	Standard	End time
Betatown	Progressive	Barcode	Minutes
Gammatown	Alternating	Standard	Minutes

Compositional Proposal (3-1-2)

3. What behavior varies? → Display output

1. Program to an interface: Strategy pattern

Introduce DisplayStrategy:

```
package paystation.domain;
/** The strategy for calculating the output for the display.
*/

public interface DisplayStrategy {
  /** return the output to present at the pay station's
      display
      @param minutes the minutes parking time
      bought so far.
   */
  public int calculateOutput( int minutes );
}
```

Refactor pay station to use DisplayStrategy:

```
  public int readDisplay() {
    return displayStrategy.calculateOutput(timeBought);
  }
```

2. Favor object composition

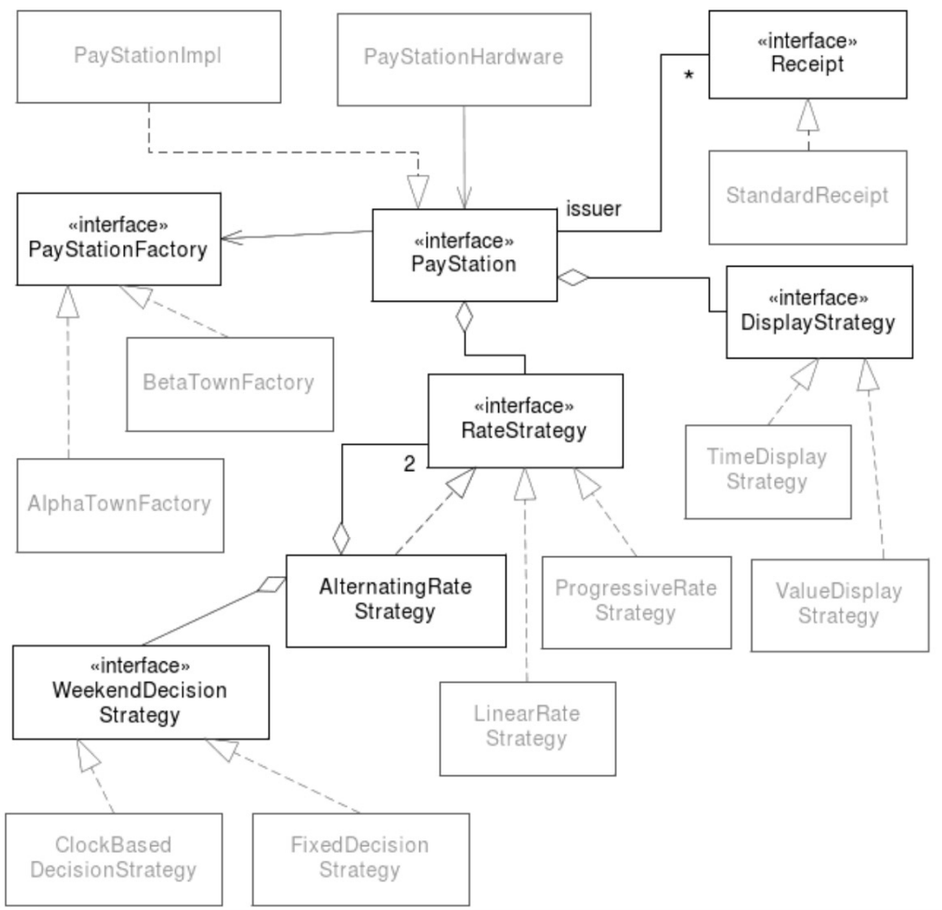
Now we can make all 12 variants by configuring the set of delegate objects that PayStationImpl should use.

Can introduce more variability through **addition**, **without modification** of existing strategies.

Maintaining Compositional Designs

The pay station compositional design is **flexible** – but is it maintainable?

- This is a complex design!
- How would we communicate this design to new developers? Make them read the textbook?



Solution: Know our patterns, and **document** them in our design

Most importantly, document central **roles** (State, Strategy, Factory)

- Concrete implementations are implied
- Developers must be trained in patterns!

Design Patterns as Roles

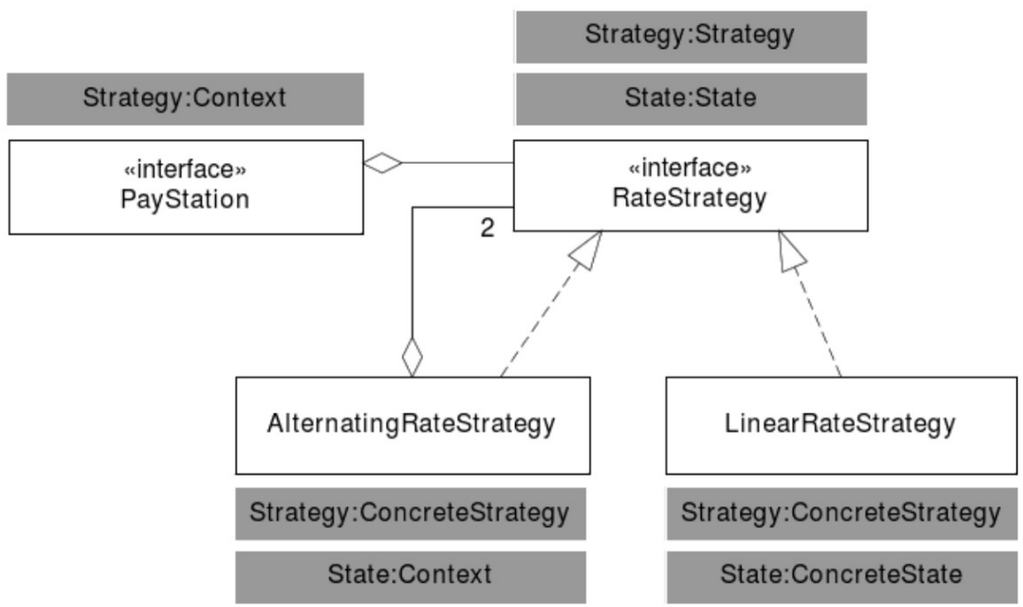
Definition: Role diagram

A role diagram is a UML class diagram in which gray boxes either above or below each interface or class describe the abstraction's role in a particular pattern. The role is described by **pattern-name:role-name**.

In pattern diagrams, think of the Strategy interface as a Strategy **role,** and the State interface as a State **role**

The **RateStrategy** interface serves the **strategy role** in the Strategy pattern, as well as the **state role** in the State pattern

A concrete instance of LinearRateStrategy may play **both** the Strategy:ConcreteStrategy and State:ConcreteState roles

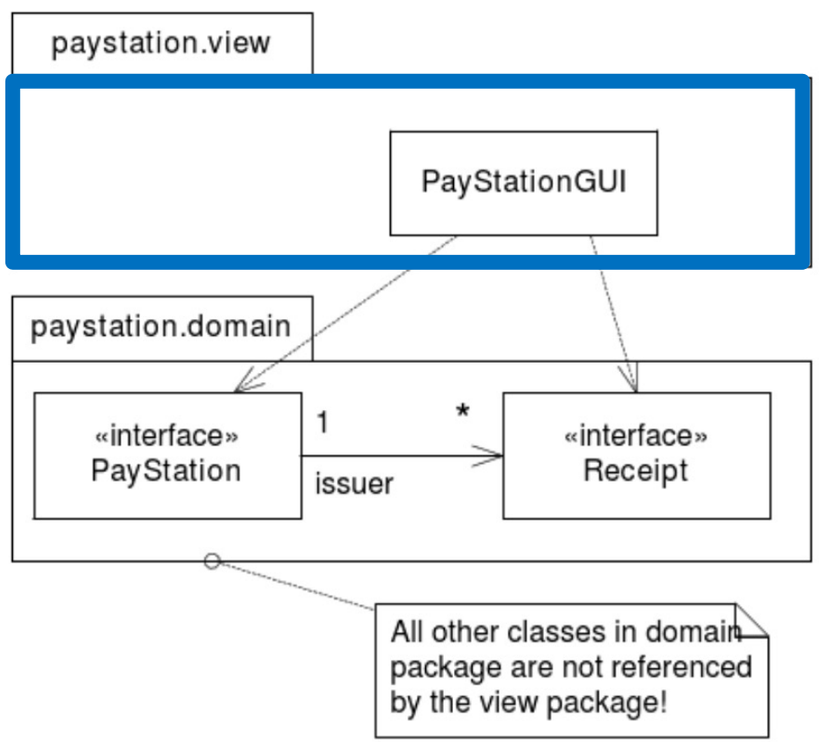


Façade Pattern

Suppose we are adding a graphical user interface (GUI) to the pay station:

paystation.view - only coupled with PayStation and Receipt

Façade: Provide a unified interface to a set of interfaces in a subsystem.



Façade Pattern

The **client** interacts with a complex subsystem, and the **façade** shields the client from the subsystem via a simple interface

- Subsystem is also shielded from changes in clients (e.g., GUI vs. hardware)
- **Weak coupling**

The pay station has embodied the façade pattern from the beginning

Pay station hardware -> PayStation interface

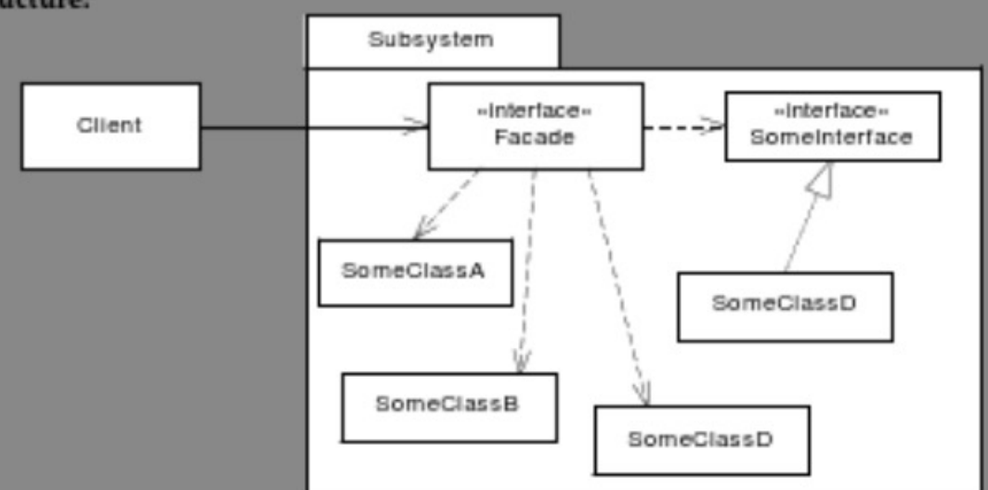
[19.1] Design Pattern: Facade

Intent Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Problem The complexity of a subsystem should not be exposed to clients.

Solution Define an interface (the facade) that provides simple access to a complex subsystem. Clients that use the facade do not have to access the subsystem objects directly.

Structure:



Roles **Facade** defines the simple interface to a subsystem. **Clients** only access the subsystem via the **Facade**.

Cost - Benefit The benefits are that it *shields clients from subsystem objects*. It *promotes weak coupling* between the client and the subsystem objects and thus lets you change these more easily without affecting the clients. A liability is that a **Facade** can bloat with a large set of methods in order for clients to access all aspects of the subsystem.

Decorator Pattern

One of AlphaTown's pay stations often overflows with 5-cent coins, so they want a **log file** of coin types with time stamps.

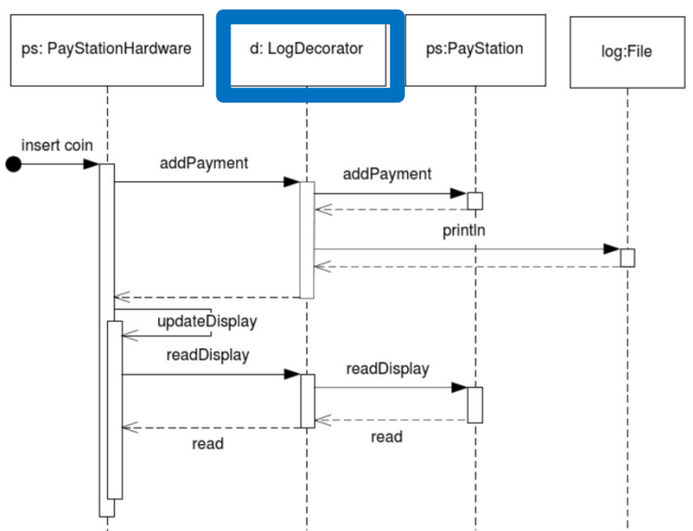
(3) What behavior varies?

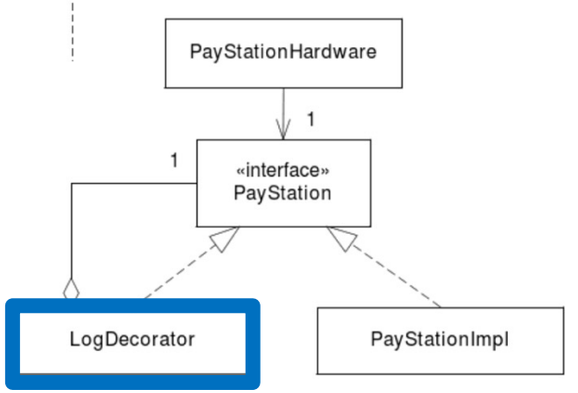
→ Accept payment (additional behavior)

(1) Program to an interface:

→ PaymentAcceptor? PayStation?

(2) We will **compose** the required behavior by putting an **intermediate object** with the same interface in front of the pay station object





Decorator: Attach additional responsibilities to an object dynamically.

Decorator Pattern

Pros:

- Composition is key: Add responsibilities to a role without affecting the underlying object
- General solution: Can decorate other implementations besides AlphaTown
- Decorators can be **chained**

Cons:

- Analyzability may suffer due to distribution of behavior across separate classes
- Delegation code in the decorator can be tedious to write

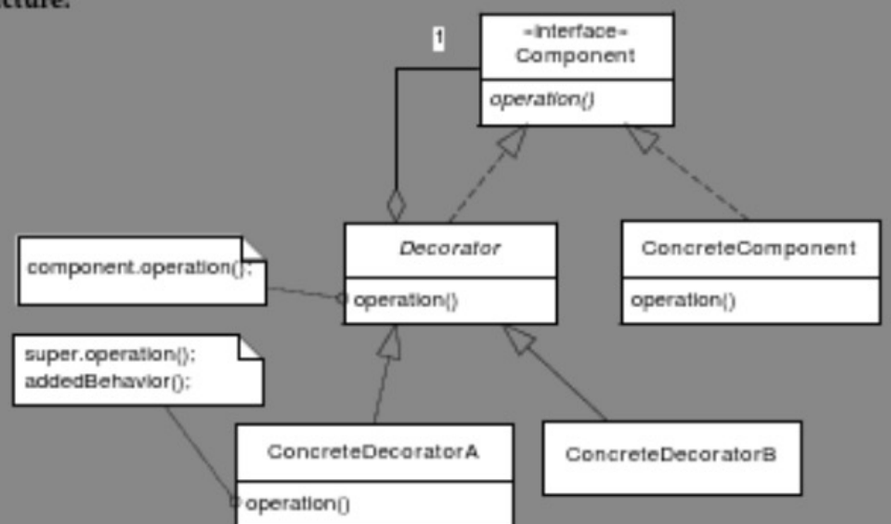
[20.1] Design Pattern: Decorator

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

Problem You want to add responsibilities and behavior to individual objects without modifying its class.

Solution You create a decorator class that responds to the same interface. The decorator forwards all requests to the decorated object but may provide additional behavior to certain requests.

Structure:



Roles **Component** defines the interface of some abstraction while **ConcreteComponent**s are implementations of it. **Decorator** defines the basic delegation code while **ConcreteDecorator**s add behavior.

Cost - Benefit Decorators allow *adding or removing responsibilities at run-time* to objects. They also allow *incrementally adding responsibilities* in your development process and thus help to keep the *number of responsibilities of decorated components low*. Decorators can provide *complex behavior by chaining* decorators after one another. A liability is that you end up with *lots of little objects* that all look alike, this can make understanding decorator chains difficult. The delegation code for each method in the decorator is a bit *tedious to write*.

Adapter Pattern

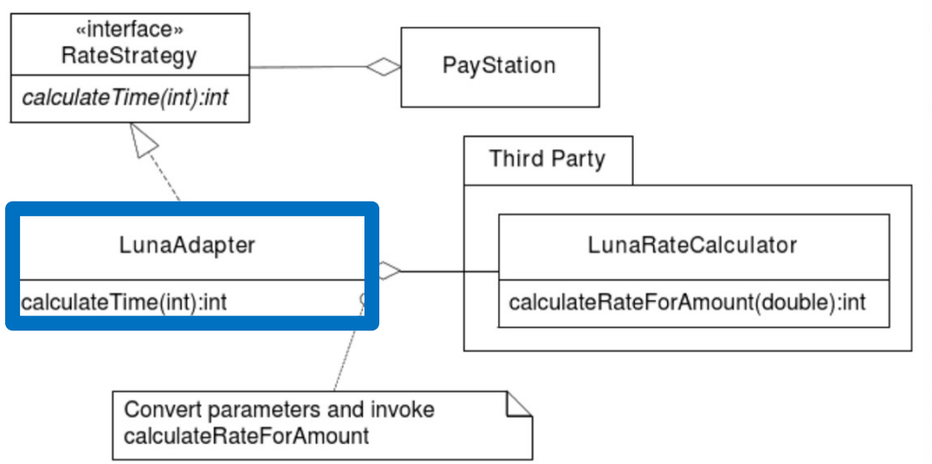
LunaTown wants to use our pay station, but they want rate correlated with **phases of the moon** (?!). They provide a class to do the calculation, but the interface does not follow our production code conventions, and we don't have access to the source code… ☹

```
public int calculateRateForAmount( double dollaramount ) {
```

(3) Variability: Rate calculation

(1) Interface: RateStrategy

(2) Composition: Provided calculator does not implement RateStrategy, so use an **intermediate object** between the pay station and LunaTown calculator



Adapter: Convert the interface of a class into another interface that clients expect.

Adapter Pattern

The adapter contains a reference to the **adaptee**, and performs parameter, protocol, and return value translations.

Adapter can adapt all implementations/subclasses of the adaptee!

But, adapter can't be reused for other adaptee classes.

[21.1] Design Pattern: Adapter

Intent Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Problem You have a class with desirable functionality but its interface and/or protocol does not match that of the client needing it.

Solution You put an intermediate object, the adapter, between the client and the class with the desired functionality. The adapter conforms to the interface used by the client and delegate actual computation to the adaptee class, potentially performing parameter, protocol, and return value translations in the process.

Structure:

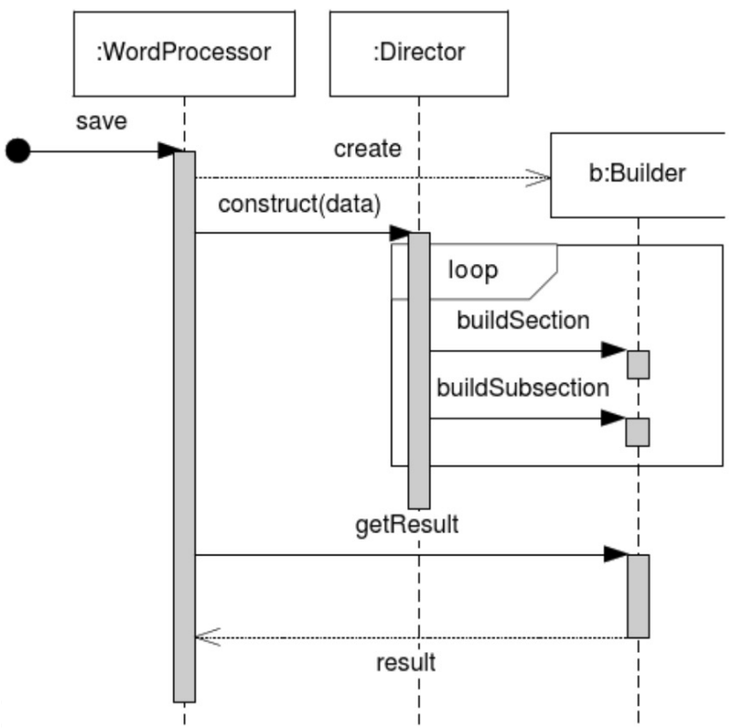
Roles **Target** encapsulates behavior used by the **Client**. The **Adapter** implements the Target role and delegate actual processing to the **Adaptee** performing parameter and protocol translations in the process.

Cost - Benefit Adapter *lets objects collaborate that otherwise are incompatible*. A single adapter can work with many adaptees—that is, all the adaptee's subclasses.

Builder Pattern

Suppose we have a word processor that stores documents with some internal data structure, and we want to export documents in different formats (e.g., XML, HTML, ASCII)

We need to avoid the **multiple maintenance problem!**



(3) Variability: Construction of output of parts like section, subsection, etc. (but the parts are common)

(1) Interface: Encapsulate the construction of parts in a **builder** interface, with methods to build each part (e.g., buildSection)

(2) Composition: Write the data structure iterator once (**director**) and have it request a delegate **builder** to construct the parts

Builder: Separate the construction of a complex object from its representation so that the same construction process can create different representations

Builder Pattern

Builder is a **creational pattern**

→Similar to Abstract Factory, but provides more control over individual elements

Pros:

- Separating construction process and concrete part building means we can use builders for other purposes
  - Favors object composition
- Easy to define new builders

Cons:

- Complex setup of construction project
- Client must know product of the builder as well as concrete builder types

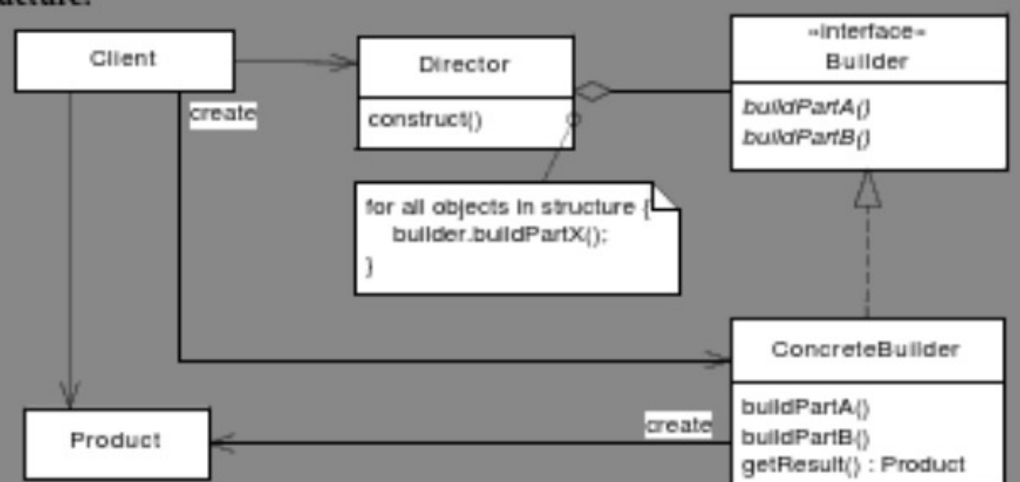
[22.1] Design Pattern: Builder

Intent Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Problem You have a single defined construction process but the output format varies.

Solution Delegate the construction of each part in the process to a builder object; define a builder object for each output format.

Structure:



Roles **Director** defines a building process but constructing the particular parts is delegated to a **Builder**. A set of **ConcreteBuilder**s is responsible to building concrete **Product**s.

Cost - Benefit It is *easy to define new products* as you can simply define a new builder. The *code for construction and representation is isolated*, and thus multiple directors can use builders and vice versa. Compared to other creational patterns (like ABSTRACT FACTORY) products are not produced in "one shot" but stepwise meaning you have *finer control over the construction process*.

Command Pattern

Suppose we want to build an application with UI elements that lets the user customize shortcut keys/buttons, define macros, and perform undo operations

(3) Encapsulate what varies: We need to handle behavior as objects that can be assigned to keys, put in macro lists, be “executable” and “un-executable” to support undo

(1) Program to an interface: ”request” objects must have a common interface to be exchanged across UI elements that enact them (**command** role, encapsulates “execute” and potentially “undo”)

(2) Favor object composition: Instead of UI elements with hardcoded behavior, they delegate to their assigned command objects

```
editor.save();
```

```
Command saveCommand = new SaveCommand(editor);
saveCommand.execute();
```

Command: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Command Pattern

Pros:

- Clients are decoupled from set of commands
- Can extend command set at runtime
- Supports multiple ways to execute a command
- Can log and store commands
  - Undo support
  - Define a macro as a composite of command objects

Cons:

- Overhead in writing and executing commands

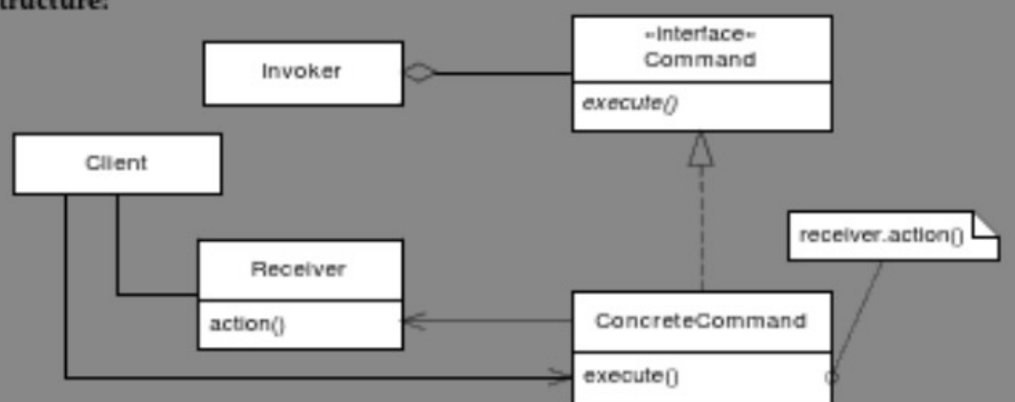
[23.1] Design Pattern: Command

Intent Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Problem You want to configure objects with behavior/actions at run-time and/or support undo.

Solution Instead of defining operations in terms of methods, define them in terms of objects implementing an interface with an execute method. This way requests can be associated to objects dynamically, stored and replayed, etc.

Structure:



Roles **Invoker** is an object, typically user interface related, that may execute a **Command**, that defines the responsibility of being an executable operation. **ConcreteCommand** defines the concrete operations that involves the object, **Receiver**, that the operation is intended to manipulate. The **Client** creates concrete commands and sets their receivers.

Cost - Benefit Objects that invoke operations are decoupled from those that know how to perform it. *Commands are first-class objects*, and can be manipulated like all other objects. You can assemble commands into *composite commands* (macros). It is *easy to add new commands*.

Proxy Pattern

Suppose we have a website or document with a lot of images

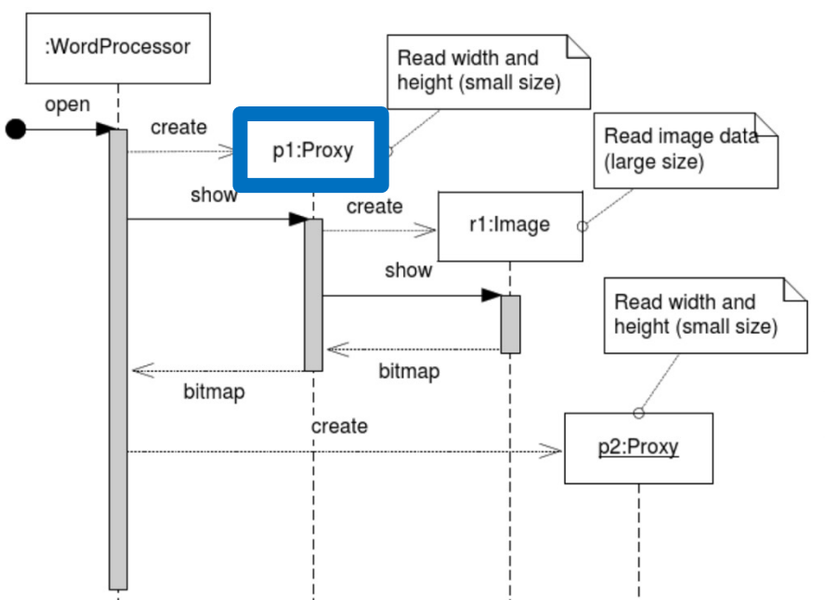
→ Loading all of the images at once will slow everything down

→ So, we want to only load the images that should currently be visible

(3) Encapsulate what varies: images that become visible will fetch image data, while those that are not visible do not load

(1) Program to an interface: provide the client with an intermediate object that will defer loading until the show() method is called on an image object

(2) Object composition: put a **proxy** object in front of the image, load essential but small data; real image is not created until show() is called



Proxy: Provide a surrogate or placeholder for another object to control access to it

Proxy Pattern

Similar benefits and liabilities as all compositional designs

Uses:

- Protection proxies / access control
- Virtual proxies / performance control
- Remote proxies / remote access

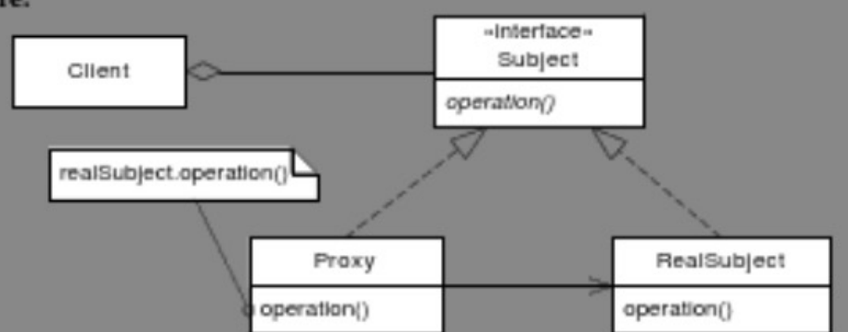
[25.1] Design Pattern: Proxy

Intent Provide a surrogate or placeholder for another object to control access to it.

Problem An object is highly resource demanding and will negatively affect the client's resource requirements even if the object is not used at all; or we need diffent types of housekeeping when clients access the object, like logging, access control, or pay-by-access.

Solution Define a placeholder object, the Proxy, that acts on behalf of the real object. The proxy can defer loading the real object, control access to it, or in other ways lower resource demands or implement housekeeping tasks.

Structure:



Roles A **Client** only interacts via a **Subject** interface. The **RealSubject** is the true object implementing resource-demanding operations (bandwidth, computation, memory usage, etc.) or operations that need access control (security, pay-by-access, logging, etc.). A **Proxy** implements the **Subject** interface and provides the relevant access control by holding a reference to the real subject and delegating operations to it.

Cost - Benefit It *strengthens reuse* as the housekeeping tasks are separated from the real subject operations. Thus the subject does not need to implement the housekeeping itself; and the proxy can act as proxy for several different types of real subjects.

Null Object Pattern

Suppose we have a class that contains some methods with long execution time, so we have a progress indicator

```
public void lengthyExecution() {
  ...
  progress.report(10);
  ..
  progress.report(50);
  ...
  progress.report(100);
  progress.end();
}
```

We don't want to bring up dialogs while testing this, so what if we set the object reference to null when testing?

Rather than absence of object, what we actually need is absence of **behavior**

→ Use a **null object**, whose methods do nothing

```
public void lengthyExecution() {
  ...
  if (progress != null )
    progress.report(10);
  ..
  if (progress != null )
    progress.report(50);
```

Null Object Pattern

Null Object contains empty implementations of all methods in the service interface

Pros:

- Ensures object reference is always valid
  - Lower risk of null pointer exceptions
  - Eliminates checks, for analyzability

Cons:

- Beware of coupling client to ConcreteService

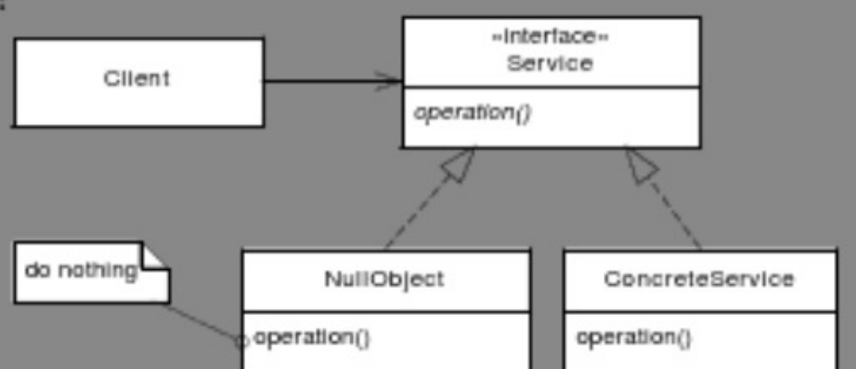
[27.1] Design Pattern: Null Object

Intent Define a no-operation object to represent null.

Problem The absence of an object, or the absence of behavior, is often represented by a reference being null but it leads to numerous checks to ensure that no method is invoked on null. It is easy to forget such checks.

Solution You create a Null Object class whose methods have no behavior, and use an instance of this class instead of using the null type. Thereby there is no need for null checking before invoking methods.

Structure:



Roles **Service** defines the interface of some abstraction while **ConcreteService** is an implementation of it. **NullObject** is an implementation whose methods do nothing.

Cost - Benefit It reduces code size and increases reliability because a lot of *testing for null is avoided*. If an interface is not already used it *requires additional refactoring to use*.

Systematic Testing

Tests can only demonstrate the presence of defects, **not the absence of defects.**

→ How can we increase the chance of finding defects?

Systematic testing is a planned and systematic process with the goal of improving the chance of finding defects while limiting the number of test cases

→ A **complement** to TDD, not an alternative

→ Costly, best for highly complex methods or systems where reliability is very important (e.g., for safety) so that the time/effort is worthwhile

Systematic Testing

Black-box testing: The UUT is treated as an opaque "black box"
→ Use the **specification** of the UUT and a **general knowledge** of programming techniques, constructs, and common programming mistakes to guide testing

White-box testing: The full implementation of the UUT is known (transparent or "white box")
→ Actual code can be inspected to generate test cases

We will focus on black-box testing

- Equivalence Class Partitioning
- Boundary Value Analysis

Equivalence Class Partitioning

An **Equivalence Class** (EC) is a subset of all possible inputs to the UUT, where if one element in the subset reveals a defect during testing, we assume that **all other elements in the subset will reveal the same defect**.

→Partition the input space into **equivalence classes,** choose a **representative** input value from each EC

Can have **valid** or **invalid** ECs

- Invalid ECs will typically define special processing, e.g., throwing an exception

Condition	Invalid ECs	Valid ECs
absolute value of x	–	$x > 0[1]$ $x \leq 0[2]$

When partitioning, aim for **coverage** and **representation:**

Coverage: Every possible input element belongs to at least one of the equivalence classes

Representation: If a defect is demonstrated by one member of an equivalence class, the same defect is assumed to be demonstrated by any other member of the class

Equivalence Class Partitioning

Often it is not easy to find equivalence classes!

- Apply heuristics, iterate, refine the ECs and test cases as you gain insight into the problem
- Take small steps

Look for conditions in the **specifications** of the UUT indicating expected input and output values

- **Set** : define an EC for each value in the set and one EC containing all elements outside the set
- **Boolean** : define one EC for the true condition and one for the false condition
- **Range** : select one valid EC that covers the allowed range and two invalid ECs, one above and one below

Equivalence Class Partitioning

With many ECs for many independent conditions, avoid combinatorial explosion:

1. Until all **valid** ECs have been covered, define a test case that covers as many **uncovered valid ECs** as possible
2. Until all **invalid** ECs have been covered, define a test case that only involves **one invalid EC**

Allowing only one condition to be invalid at a time avoids **masking**

ECs covered	Test case	Expected output
[1], [6]	(' ',5)	illegal
[2], [6]	('j',3)	illegal
[3], [4]	('b',0)	illegal
[3], [5]	('c',9)	illegal
[3], [6]	('b',6)	legal

One valid test case
Only invalid row or column (not both),
test lower and higher than allowed range

Equivalence Class Partitioning

1. Review the requirements for the UUT, identify **conditions,** and use heuristics to find ECs for each condition
   - Create an **equivalence class table**
2. Review the ECs and consider the **representation** of elements in each EC. Repartition the EC if elements are not representative.
3. Review to verify **coverage**
4. Generate test cases from the ECs. Apply heuristics to generate a minimal set of test cases.
   - Create a **test case table**
5. Review test cases to find what is missing, **iterate**!

Boundary Value Analysis

Boundary value: an element that lies right on or next to the edge of an equivalence class

→Not always applicable, depends on the requirements and implementation

Example: Check for a valid chess board position

```
if ( row <= 1 ) return false; // should have been row < 1
```

Set and Boolean conditions generate ECs where boundaries are **part of the ECs themselves**

Code Structure

Generally a program is a combination of three types of structures:

- **Sequences**: blocks of code, sequential operations
- **Selections**: decisions, conditions, handled with if, switch, etc.
- **Iterations:** repetitions, implemented with loops (for, while, etc.)

If we know the structures contained in a program (white-box), we can evaluate the ability of our test suites to **exercise** these structures.

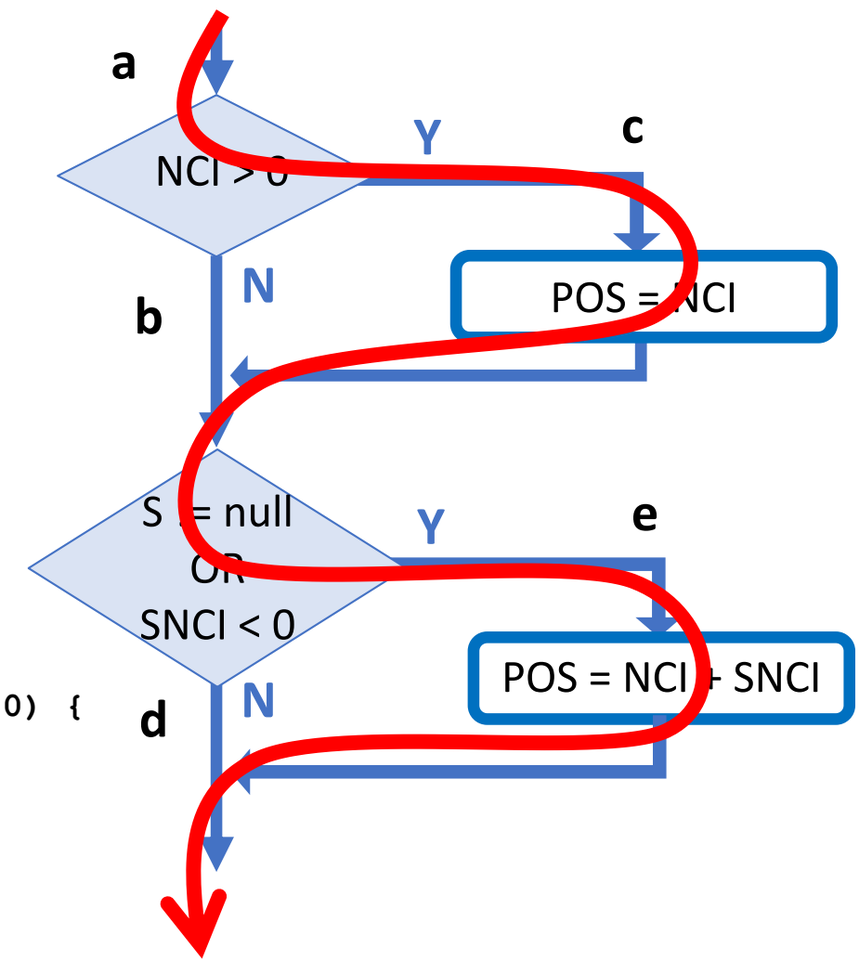
Test Coverage

Statement coverage: require every statement of the program to be executed at least once

Test Case: NCI = 1000, SNCI = -2000
Expected taxationBasis 0, result is -1000
→ Defect detected!

A defect remains!
S == null will cause an error in the check SNCI < 0

```
calculateBracket(Taxpayer t) {
    int posNetCapitalIncome = 0;
    if (t.netCapitalIncome() > 0) {
        posNetCapitalIncome = t.netCapitalIncome();
    }
    if (t.getSpouse() != null || t.getSpouse().netCapitalIncome() < 0) {
        posNetCapitalIncome = posNetCapitalIncome +
            t.getSpouse().netCapitalIncome();
    }
    int taxationBasis = t.personalIncome() + posNetCapitalIncome;
```



Test Coverage

Decision (branch) coverage: require each condition has a true and false outcome at least once

Test Case 1: NCI = 1000, SNCI = -2000

Test Case 2: NCI = -2000, S == null

Expected taxationBasis = 0, result is null exception

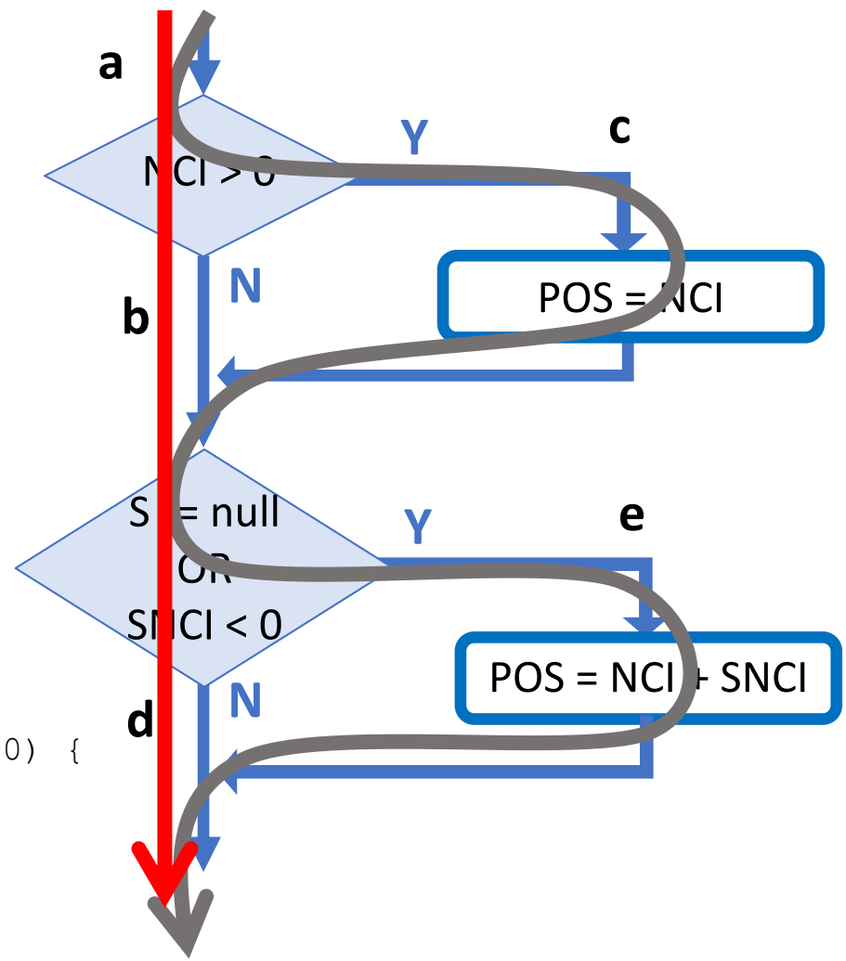
→ Defect detected!

A defect remains!

NCI < 0; SNCI < 0 path is untested

expect taxationBasis 0, result is negative

```
calculateBracket(Taxpayer t) {
    int posNetCapitalIncome = 0;
    if (t.netCapitalIncome() > 0) {
        posNetCapitalIncome = t.netCapitalIncome();
    }
    if (t.getSpouse() != null && t.getSpouse().netCapitalIncome() < 0) {
        posNetCapitalIncome = posNetCapitalIncome +
            t.getSpouse().netCapitalIncome();
    }
    int taxationBasis = t.personalIncome() + posNetCapitalIncome;
```



Test Coverage

- Condition/Decision coverage: test every condition and decision outcome
- Multiple-condition coverage: test all condition combinations in a decision
- **Path coverage: test every possible combination of decisions**
- Others… (exceptions?)

Complex (and sometimes impractical) to compute and make test cases for, not handled by most code coverage tools.

Becomes more important for safety-critical systems
A Practical Tutorial on Modified Condition/Decision Coverage (NASA)

Decision coverage usually satisfies statement coverage.

100% line/branch coverage is not necessary, but low coverage is worrisome.

100% line/branch coverage does not guarantee no defects!

→ Consider use cases, common configurations to prioritize paths

Composite Pattern

Suppose that in a graphics editor we have several graphical objects that need to be moved, resized, etc. as a group.

→Group may be part of a larger group

→Hierarchical, “part-whole” structure

1. Program to an interface:

→Define a common interface for the **part** and the **whole**

2. Compose required behavior (recursively)

Composite: Compose objects into tree structures to represent part-whole hierarchies, such that clients treat individual objects and compositions of objects uniformly.

```
/** Define the Component interface
 * (partial for a folder hierarchy) */
interface Component {
  public void addComponent(Component child);
  public int size();
}
```

```
/** Define a (partial) folder abstraction */
class Folder implements Component {
  private List<Component> components = new ArrayList<Component>();
  public void addComponent(Component child) {
    components.add(child);
  }
  public int size() {
    int size = 0;
    for ( Component c: components ) {
      size += c.size();
    }
    return size;
  }
}
```

Lecture 19

Composite Pattern

Pros:

- Common interface for leaf and composite
  - Enables an abstract class defining operations that are identical for both

Cons:

- Composite trades lower cohesion of the leaf to get a uniform interface for all objects of the part-whole structure

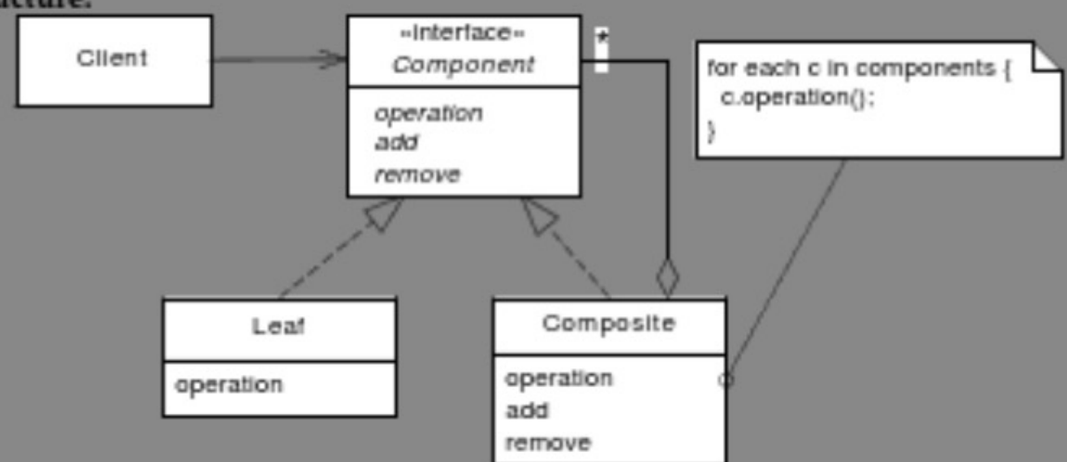
[26.1] Design Pattern: Composite

Intent Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Problem Handling of tree data structures.

Solution Define a common interface for composite and atomic components alike. Define composites in terms of a set of children, each either a composite or atomic component. Define composite behavior in terms of aggregating or composing behavior of each child.

Structure:



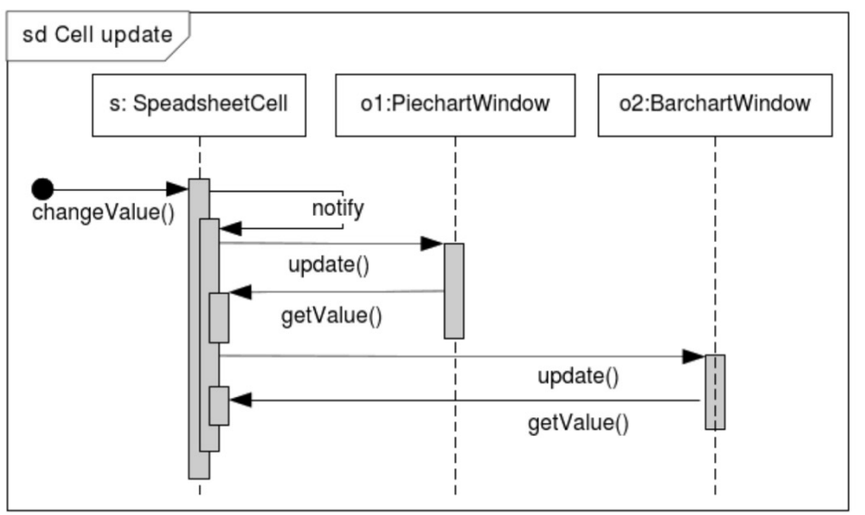
Roles **Component** defines a common interface. **Composite** defines a component by means of aggregating other components. **Leaf** defines a primitive, atomic, component i.e. one that has no substructure.

Cost - Benefit It defines a *hierarchy of primitive and composite objects.* It makes the *client interface uniform* as it does not need to know if it is a simple or composite component. It is *easy to add new kinds of components* as they will automatically work with the existing components. A liability is that the *design can become overly general* as it is difficult to constrain the types of leafs a composite may contain. The *interfaces may method bloat* with methods that are irrelevant; for instance an add method in a leaf.

Observer

Consider a spreadsheet where we want to simultaneously display the same data with a bar chart and a pie chart in their own windows

→The windows should be synchronized and updated (redrawn) according to changes in the data



3. Variability: When the subject's state changes, we know the observers have to do some processing, but we don't know what kind of processing.
1. Program to an interface: **Observer** interface for the processing responsibility
2. Composition: **Subject** maintains a set of observers and is responsible for invoking the update method of all its observers

Observer: Define a dependency among objects so that when one object changes state, all dependents are notified and updated

Lecture 19

Observer Pattern

Pros:

- Loose coupling between subject and observer
- Can have many subjects, many observers

Cons:

- Every state change will trigger the notification protocol

  "Flickering" when redrawing application graphics

- Risk of circular dependencies

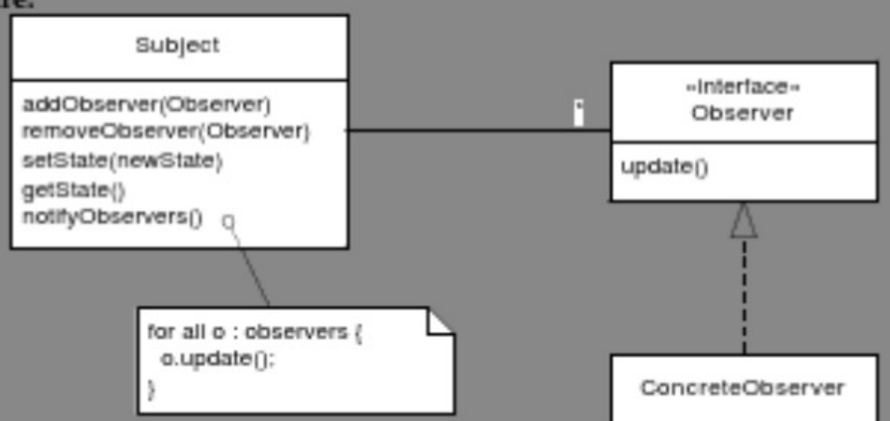
[28.1] Design Pattern: Observer

Intent Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Problem A set of objects needs to be notified in case a common object changes state to ensure system wide consensus and consistency. You want to ensure this consistency in a loosely coupled way.

Solution All objects that must be notified (**Observers**) implements an interface containing an update method. The common object (**Subject**) maintains a list of all observers and when it changes state, it invokes the update method on each object in the list. Thereby all observing objects are notified of state changes.

Structure:



Roles **Observer** specifies the responsibility and interface for being able to be notified. **Subject** is responsible for holding state information, for maintaining a list of all observers, and for invoking the update method on all observers in its list. **ConcreteObserver** defines concrete behavior for how to react when the subject experiences a state change.

Cost - Benefit The benefits are: *Loose coupling between Subject and Observer* thus it is easy to add new observers to the system. *Support broadcast communication* as it is basically publishing information in a one-to-many relation. The liabilities are: *Unexpected/multiple updates:* as the coupling is low it is difficult for observers to infer the nature of subject state changes which may lead to the observers updating too often or spuriously.

Model-View-Controller

Vector graphics editors (e.g., Inkscape, Visio) present a palette of shapes that a user can add to a drawing and then manipulate with the mouse. The drawing can be viewed in multiple windows simultaneously showing different parts at different scales.

The drawing needs to be rendered in the windows simultaneously
→ **Observer** pattern: update windows when state changes

The drawing needs to process user inputs from multiple sources
→ Response to mouse click input should change depending on the type of object that was clicked
→ **State** pattern: alter behavior when the state changes

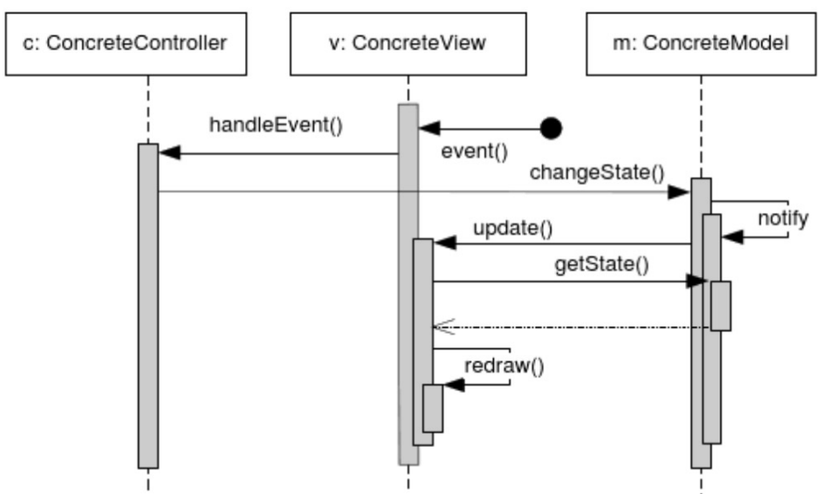


Figure 29.3: MVC protocol.

Model-View-Controller: define a loosely coupled design to form the architecture of graphical user interfaces with multiple windows and user input from multiple input sources.

Lecture 19

Model-View-Controller Pattern

MVC is an **architectural** pattern

Similar pros/cons as observer and state

Structure and protocol are complex, must be programmed with care

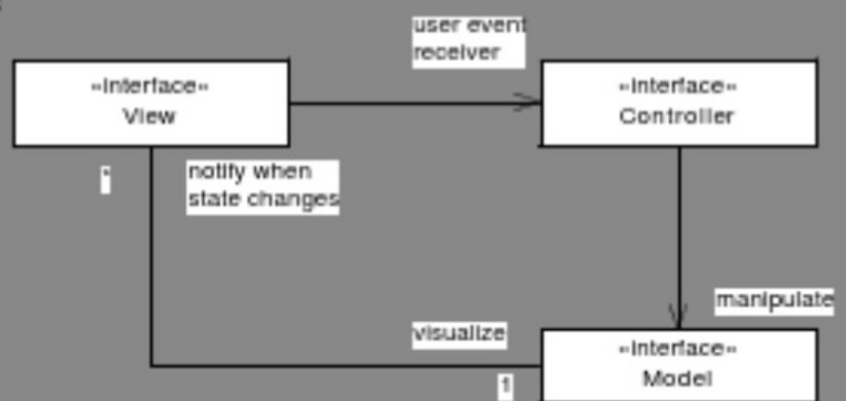
[29.1] Design Pattern: Model-View-Controller

Intent Define a loosely coupled design to form the architecture of graphical user interfaces having multiple windows and handling user input from mouse, keyboard, or other input sources.

Problem A graphical user interface must support multiple windows rendering different visual representations of an underlying set of state-full objects in a consistent way. The user must be able to manipulate the objects' state using mouse and keyboard.

Solution A **Model** contains the application's state and notifies all **Views** when state changes happen. The **Views** are responsible for rendering the model when notified. User input events are received by a **View** but forwarded to its associated **Controller**. The **Controller** interprets events and makes the appropriate calls to the **Model**.

Structure:



Roles **Model** maintains application state and updates all associated **Views**. **View** renders the model graphically and delegates user events to the **Controller** that in turn is responsible for modifying the model.

Cost - Benefit The benefits are: *loose coupling between all three roles* meaning you can add new graphical renderings or user event processing. *Multiple views/windows* are supported. It is possible to *change event processing at run-time*. The liabilities are: *unexpected/multiple updates:* as the coupling is low it is difficult for views to infer the nature of model state changes which may lead to graphical flickering. *Design complexity* is another concern if the development team is untrained.

Template Method

Consider the pay station algorithm to receive payment:

1. Check that input is a valid coin
2. Add coin value to payment sum
3. Calculate minutes of parking from payment sum

There is a strict **sequence** of steps required for the algorithm to be correct.

→What if we want to change behavior of the individual steps without changing the overall algorithm sequence?

→Sounds like a job for Strategy! (e.g., variability in step 3)

Template Method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses or delegates, let behavior of certain steps of an algorithm be varied without changing the algorithm structure.

Template Method

Unification Variant

```
abstract class AbstractClass {
  public void templateMethod() {
    [fixed code part 1]
    step1();
    [fixed code part 2]
    step2();
    [fixed code part 3]
  }
  protected abstract void step1();
  protected abstract void step2();
}
class ConcreteClass extends AbstractClass() {
  protected void step1() {
    [step 1 specific behavior]
  }
  protected void step2() {
    [step 2 specific behavior]
  }
}
```

The Template Method pattern is focused on method abstractions

Separation Variant

```
class Class {
  private HookInterface1 hook1;
  private HookInterface2 hook2;
  public void setHook( HookInterface1 hook1,
                       HookInterface2 hook2) {
    this.hook1 = hook1;
    this.hook2 = hook2;
  }
  public void templateMethod() {
    [fixed code part 1]
    hook1.step1();
    [fixed code part 2]
    hook2.step2();
    [fixed code part 3]
  }
}
interface HookInterface1 {
  public void step1();
}
interface HookInterface2 {
  public void step2();
}
class ConcreteHook1 implements HookInterface1() {
  public void step1() {
    [step 1 specific behavior]
  }
class ConcreteHook2 implements HookInterface2() {
  public void step2() {
    [step 2 specific behavior]
  }
}
```

Lecture 19

Template Method Pattern

Pros:

- Avoids multiple maintenance
- Hook behavior can be easily changed
- Hook methods can be varied independently

Cons:

- Added complexity
- Added work to encapsulate steps

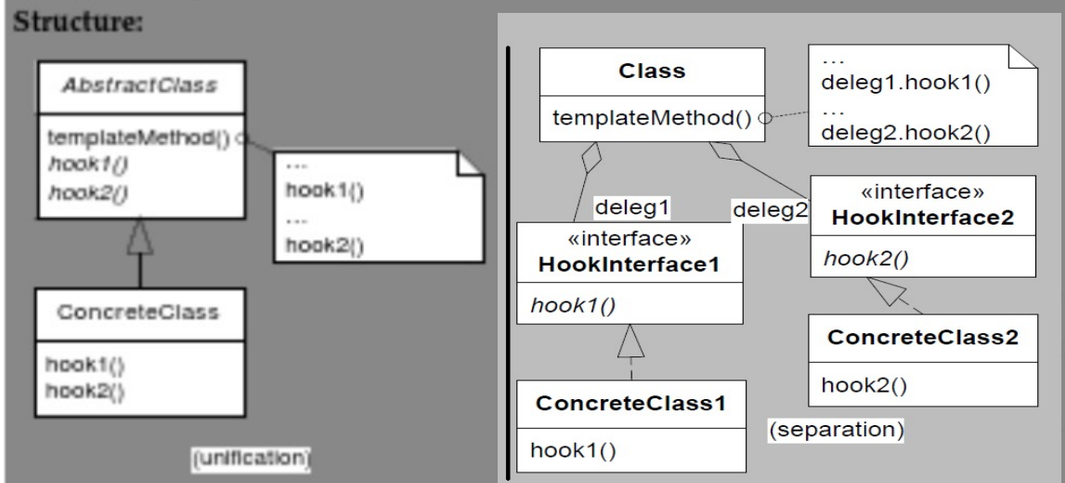
[31.1] Design Pattern: Template Method

Intent Define the skeleton of an algorithm in an operation, deferring some steps to subclasses or delegates. Template Method lets the behavior of certain steps of an algorithm be varied without changing the algorithm's structure.

Problem There is a need to have different behaviors of some steps of an algorithm but the algorithm's structure is otherwise fixed.

Solution Define the algorithm's structure and invariant behavior in a template method and let it call hook methods that encapsulate the steps with variable behavior. Hook methods may either be abstract methods in the same class as the template method, or they may be called on delegate object(s) implementing one or several interfaces defining the hook methods. The former variant is the *unification* variant, the latter the *separation* variant.

Structure:



Roles The roles are method abstractions: the **template method** defines the algorithm structure and invariant behavior. **Hook methods** encapsulate variable behavior. The **HookInterface** interface defines the method signatures of the hook methods.

Cost - Benefit The benefits are that the *algorithm template is reused* and thus avoids multiple maintenance; that the *behavior of individual steps, the hooks, may be changed*; and that *groups of hook methods may be varied independently* (separation variant only). The liability is *added complexity of the algorithm* as steps have to be encapsulated.

Characteristics of a Framework

Skeleton / design / high-level language / template

→Application at a high level of abstraction; a basic conceptual structure

Application / class of software / within a domain

→Behavior in a well-defined domain

Cooperating / collaborating classes

→Defined protocol for a set of well-defined components/objects; user must understand the protocol(s) and program accordingly

Customize / abstract classes / reusable / specialize

→Flexibility; can be tailored to a concrete context (within the domain)

Classes / implementation

→ Reuse of working code as well as reuse of design

Frameworks: Examples

Pay Station framework (evolved from AlphaTown Pay Station application)

- A skeleton within a particular domain
- Collaborating classes
- Classes customized for a particular product variant
- Implementation / code reuse as well as design

HotCIv

- Framework supports different game variants

MiniDraw

- A framework that supports user interaction with 2D image-based graphics via mouse events
- Will use for HotCiv GUI

Frameworks: Customization

Key point: Frameworks are not customized by code modification!

- Source code must not be altered, even if accessible
- Customize through mechanisms provided by framework developers

→**Change by addition**

Recall: Template Method

Frozen spots: Parts of the framework code that cannot be altered; define the basic design and protocols

Hot spots / hook methods / variability points: Clearly defined parts of the framework in which specialization code can alter or add behavior to the final application

→ Subclassing or delegation

Frameworks: Customization

Key point: Frameworks must use dependency injection!

- Enables customization by the application developer
- Objects that define variability points (hot spots) are instantiated by the **application** code and **injected** into the framework

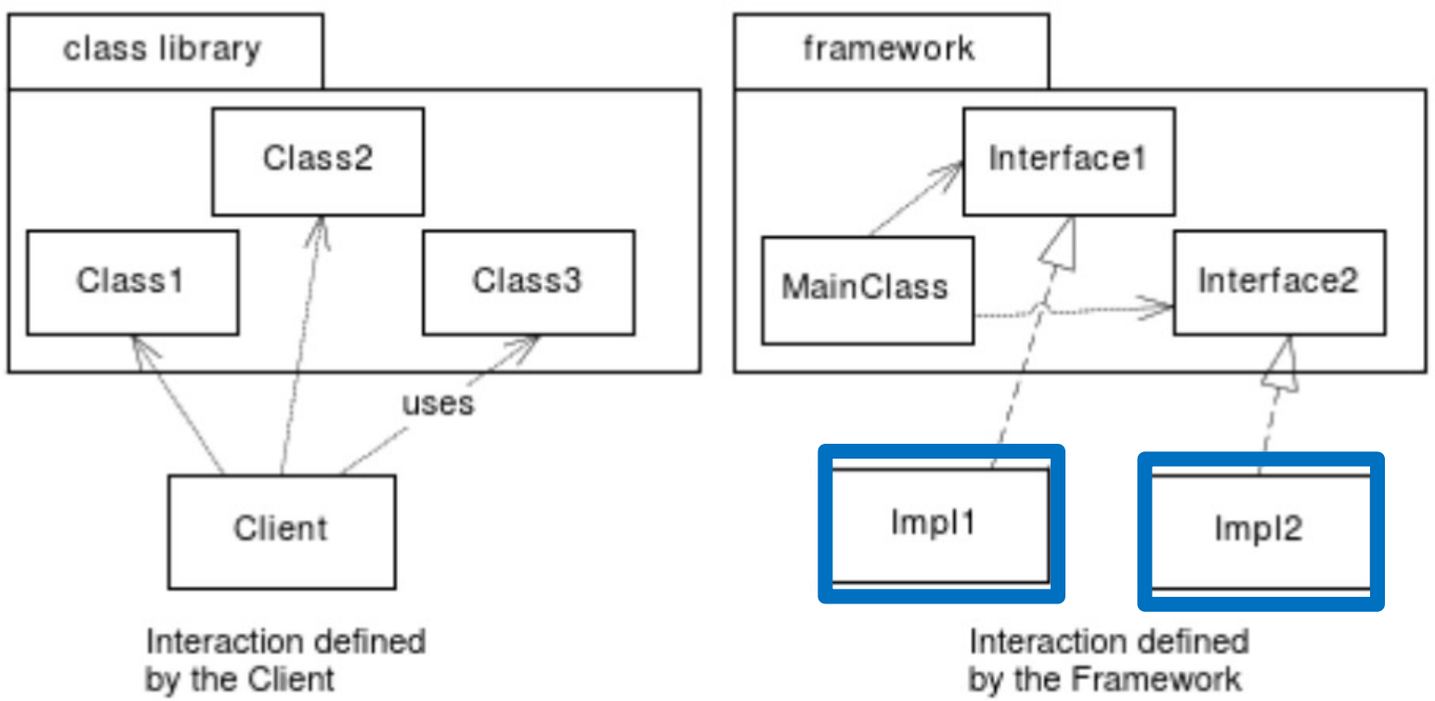
Key point: Frameworks should provide a spectrum of no implementation (interface) to partial implementation (abstract) to full implementation for variability points

Existing concrete classes: predefined classes in the framework	**Subclass abstract class:** the framework contains abstract classes	**Implement an interface**: the framework contains an interface

Frameworks: Inversion of Control

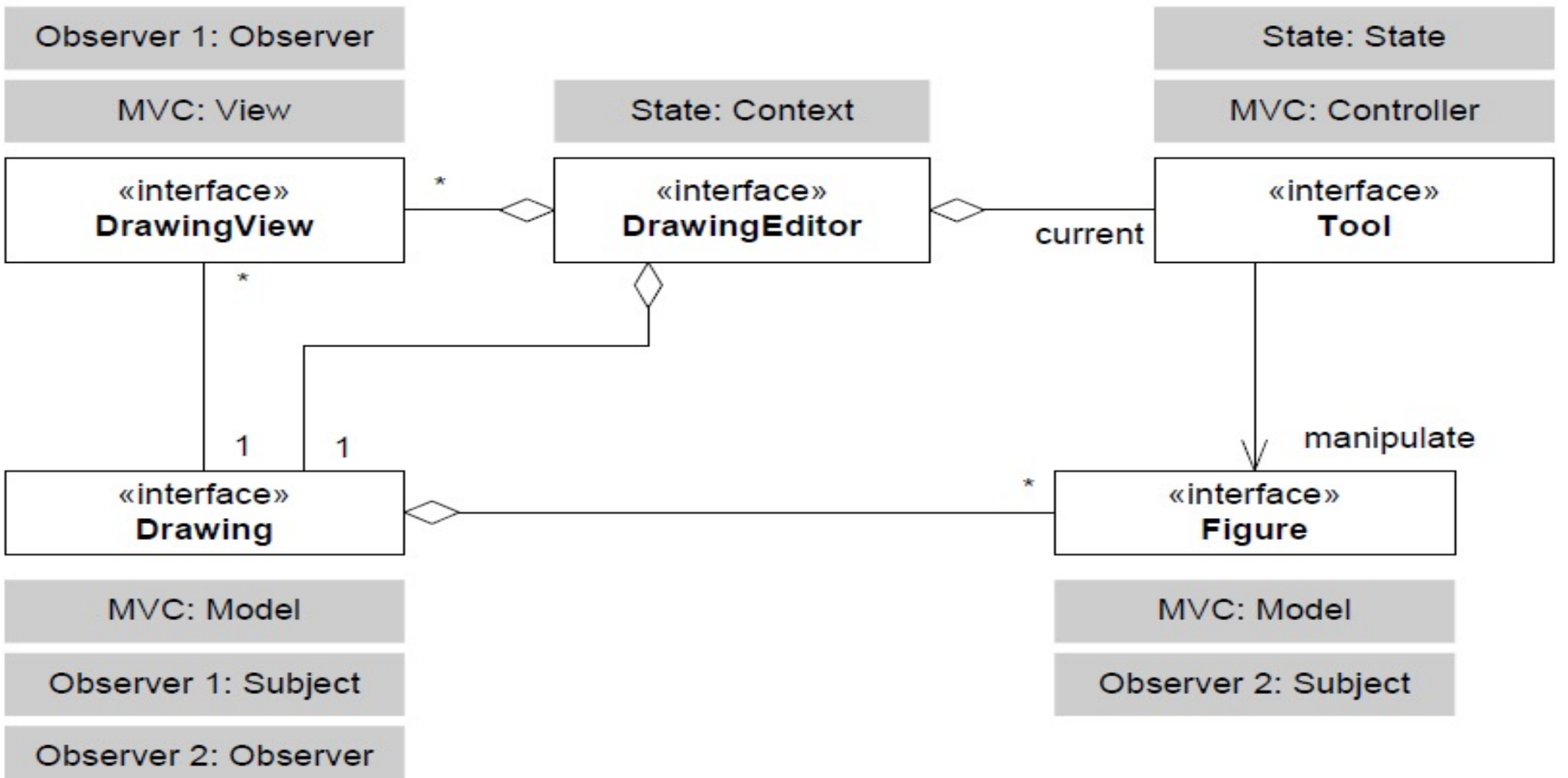
The **framework** (not the application developer) defines the **flow of control** in an application.

→ Contrast with libraries, where the application maintains control



Hot spot objects

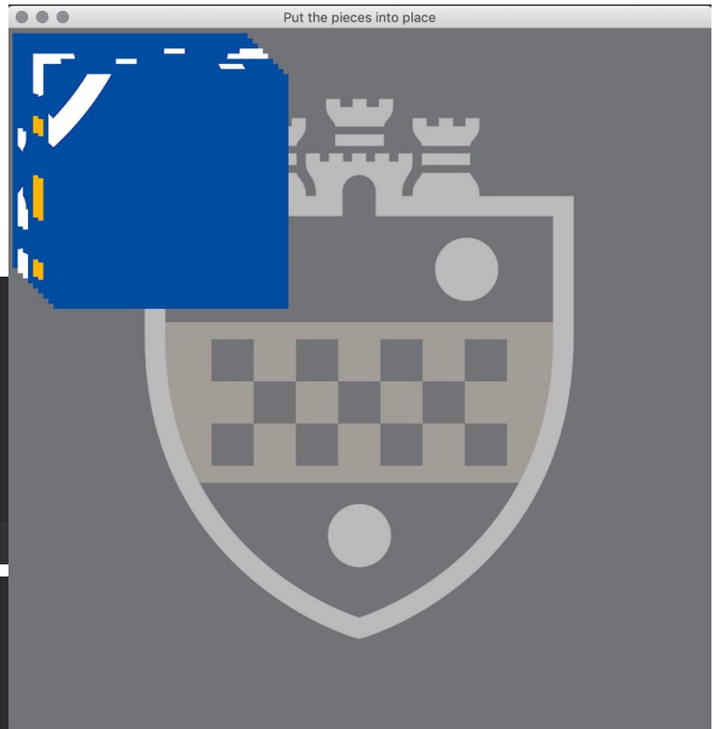
MiniDraw


Observer 1: Observer
State: State
MVC: View
State: Context
MVC: Controller
«interface»
DrawingView
*
«interface»
DrawingEditor
current
«interface»
Tool
*
1
1
manipulate
*
«interface»
Drawing
«interface»
Figure
MVC: Model
MVC: Model
Observer 1: Subject
Observer 2: Subject
Observer 2: Observer

MiniDraw: Puzzle

```
1   package puzzle;
2   import minidraw.standard.*;
3   import minidraw.framework.*;
4   import java.awt.*;
5   import javax.swing.*;
6
```

```
34  public class LogoPuzzle {
35
36    public static void main(String[] args) {
37      DrawingEditor editor =
38        new MiniDrawApplication( title: "Put the pieces into place",
39                                 new PuzzleFactory() );
40      editor.open();
41      editor.setTool( new SelectionTool(editor) );
42
43      Drawing drawing = editor.drawing();
44      drawing.add(  new ImageFigure( imagename: "11", new Point( x: 5,  y: 5)) );
45      drawing.add(  new ImageFigure( imagename: "12", new Point( x: 10,  y: 10)) );
46      drawing.add(  new ImageFigure( imagename: "13", new Point( x: 15,  y: 15)) );
47      drawing.add(  new ImageFigure( imagename: "21", new Point( x: 20,  y: 20)) );
48      drawing.add(  new ImageFigure( imagename: "22", new Point( x: 25,  y: 25)) );
49      drawing.add(  new ImageFigure( imagename: "23", new Point( x: 30,  y: 30)) );
50      drawing.add(  new ImageFigure( imagename: "31", new Point( x: 35,  y: 35)) );
51      drawing.add(  new ImageFigure( imagename: "32", new Point( x: 40,  y: 40)) );
52      drawing.add(  new ImageFigure( imagename: "33", new Point( x: 45,  y: 45)) );
53    }
54  }
```



Instantiate DrawingEditor

Open DrawingEditor (open app window)

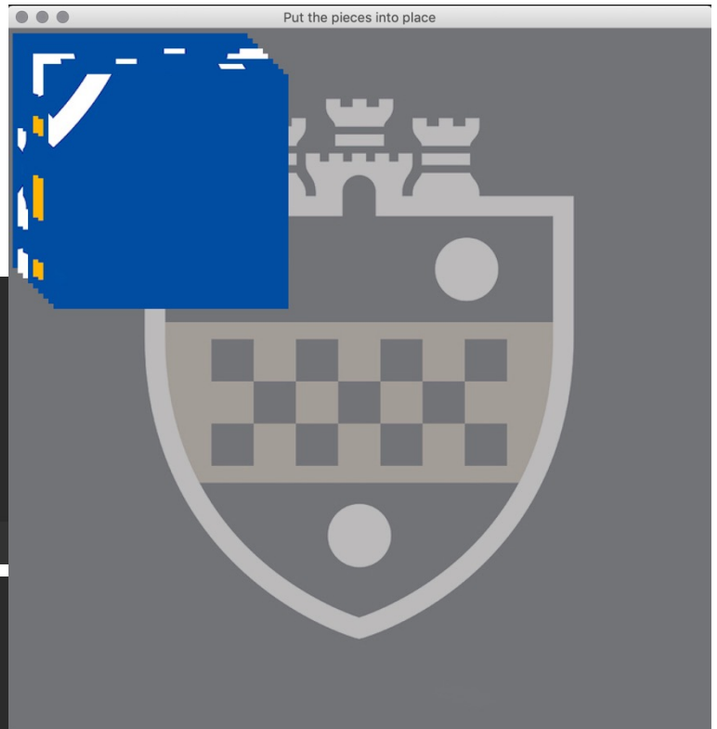
Set Tool

→ Template

MiniDraw: Puzzle

```
1   package puzzle;
2   import minidraw.standard.*;
3   import minidraw.framework.*;
4   import java.awt.*;
5   import javax.swing.*;
6
```

```
34  public class LogoPuzzle {
35
36    public static void main(String[] args) {
37      DrawingEditor editor =
38        new MiniDrawApplication( title: "Put the pieces into place",
39                                 new PuzzleFactory() );
40      editor.open();
41      editor.setTool( new SelectionTool(editor) );
42
43      Drawing drawing = editor.drawing();
44      drawing.add(  new ImageFigure( imagename: "11", new Point( x: 5,  y: 5)) );
45      drawing.add(  new ImageFigure( imagename: "12", new Point( x: 10,  y: 10)) );
46      drawing.add(  new ImageFigure( imagename: "13", new Point( x: 15,  y: 15)) );
47      drawing.add(  new ImageFigure( imagename: "21", new Point( x: 20,  y: 20)) );
48      drawing.add(  new ImageFigure( imagename: "22", new Point( x: 25,  y: 25)) );
49      drawing.add(  new ImageFigure( imagename: "23", new Point( x: 30,  y: 30)) );
50      drawing.add(  new ImageFigure( imagename: "31", new Point( x: 35,  y: 35)) );
51      drawing.add(  new ImageFigure( imagename: "32", new Point( x: 40,  y: 40)) );
52      drawing.add(  new ImageFigure( imagename: "33", new Point( x: 45,  y: 45)) );
53    }
54  }
```



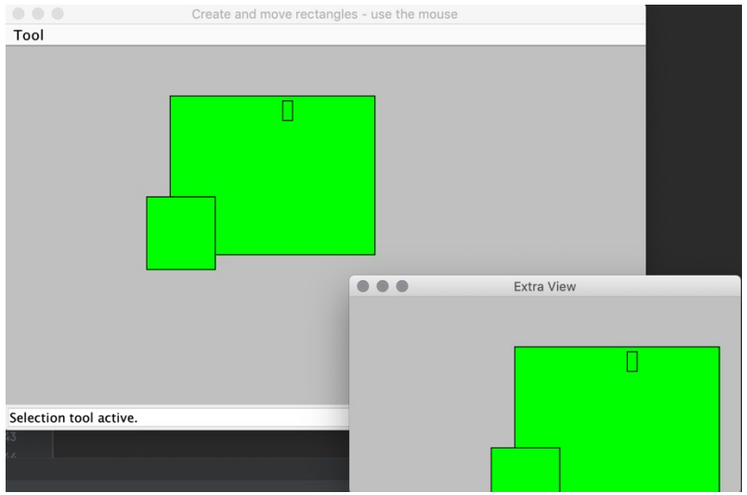
Factory: Concrete implementations of DrawingView and Drawing (and optional status text field)

Drawing: Collection of figures
Once added, they show up in the window

Tool: Manipulate the drawing area
SelectionTool: Move and selection behavior

MiniDraw: Rectangles

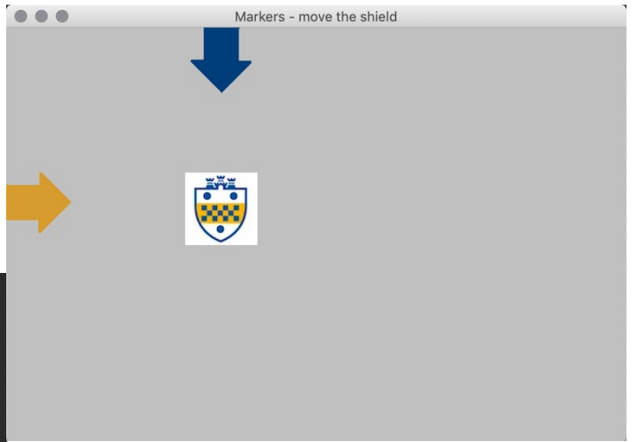
```
42  public class ShowRectangle {
43
44    public static void main(String[] args) {
45      Factory f = new EmptyCanvasFactory();
46      DrawingEditor editor =
47        new MiniDrawApplication( title: "Create and move rectangles "+
48                                 "- use the mouse", f );
49      Tool
50        rectangleDrawTool = new RectangleTool(editor),
51        selectionTool = new SelectionTool(editor);
52      addToolSelectMenusToWindow( editor,
53                                  rectangleDrawTool,
54                                  selectionTool );
55      editor.open();
56
57      editor.setTool( rectangleDrawTool );
58      editor.showStatus( "MiniDraw version: "+DrawingEditor.VERSION );
59
60      // create second view
61      JFrame newWindow = new JFrame( title: "Extra View");
62      newWindow.setLocation( x: 620, y: 20 );
63      newWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
64
65      DrawingView extraView = f.createDrawingView(editor);
66      JPanel panel = (JPanel) extraView;
67      newWindow.getContentPane().add(panel);
68      newWindow.pack();
69      newWindow.setVisible(true);
70    }
```



RectangleTool: Draws rectangles
SelectionTool: Select/move rectangles
Select tool in menu
→ State

Second window contains second view
Views are synchronized
→ Observer

MiniDraw: Markers




```
46  public class Markers {
47    public static void main(String[] args) {
48      Factory f = new EmptyCanvasFactory();
49      DrawingEditor editor =
50        new MiniDrawApplication( title: "Markers - move the shield", f );
51
52      editor.open();
53      Figure logo = new ImageFigure( imagename: "pitt-shield-small", new Point( x: 200,  y: 200));
54
55      Figure rightArrow =
56        new MarkerFigureDecorator( new ImageFigure( imagename: "arrow-right",
57                                                    new Point( x: 0,  y: 200)),
58                                   logo,
59                                   horizontal: false );
60      Figure downArrow =
61        new MarkerFigureDecorator( new ImageFigure( imagename: "arrow-down",
62                                                    new Point( x: 200,  y: 0)),
63                                   logo,
64                                   horizontal: true );
65
66      editor.setTool( new SelectionTool(editor) );
67
68      editor.drawing().add(rightArrow);
69      editor.drawing().add(downArrow);
70      editor.drawing().add(logo);
71    }
72  }
```

MarkerFigureDecorator:

Registers arrow figures as observers of changes in logo figure that move in response to logo position changes

MiniDraw: Variability Points

Images

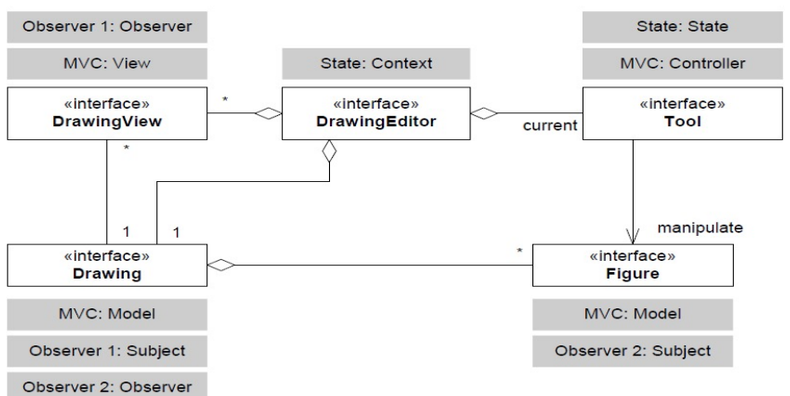
→Custom images, loaded from a specific folder (src/main/resources/minidraw-images)

Tools

→Tell the editor which tool to use to handle figures in different ways

Figures

→Custom Figures that do their own graphical rendering



Views

→Custom DrawingViews that provide special rendering

Drawings

→Customize collection implementation to store and remove figures

Observers

→ Make other objects listen to state changes in figures (including other figures)

Error Handling: Types of Errors

1. Problems with external data or conditions
   - Notify the user/client, don't crash
2. Internal errors (bad arguments, out of memory, unexpected result from a function call)
   - Debug: Notify programmer, then crash
   - Production: Don't crash, recover gracefully if possible

Error Handling: Severity

Fatal errors

- Cannot continue execution (or it would be meaningless to)
  - e.g., out of memory

Nonfatal (for now)

- Potentially fatal later
- Start recovery strategy, inform user/client before it is too late

Nonfatal

- Recovery is possible, may want to inform anyway

Error Handling: More Guidelines

Depending on type and severity:

- Catch if possible
- Handle specifically but systematically
- Log (System.out, logging framework, file…)

Even before that:

- Anticipate likely errors, try to avoid them in the first place
- Weigh the risk of not handling vs. cost of handling exceptions
  - Sometimes better to abort
  - Don’t ignore or assume errors won’t happen (without good cause)

Defensive Programming

Assertions: Check that everything is operating as expected (error if not)

- Document assumptions made in code, pre/post-conditions
- Intended to be silent, things that should never occur

Exceptions: Notify other parts of the program about errors that should not be ignored

- Only for conditions that are really “exceptional”
- If it could be ignored, use a code instead
- If possible to handle locally, do that

Don’t throw errors in constructors and destructors!

Throw at the right level of abstraction

- Don’t expose implementation details
- Catch exceptions from lower levels and translate

Error Handling: FMEA

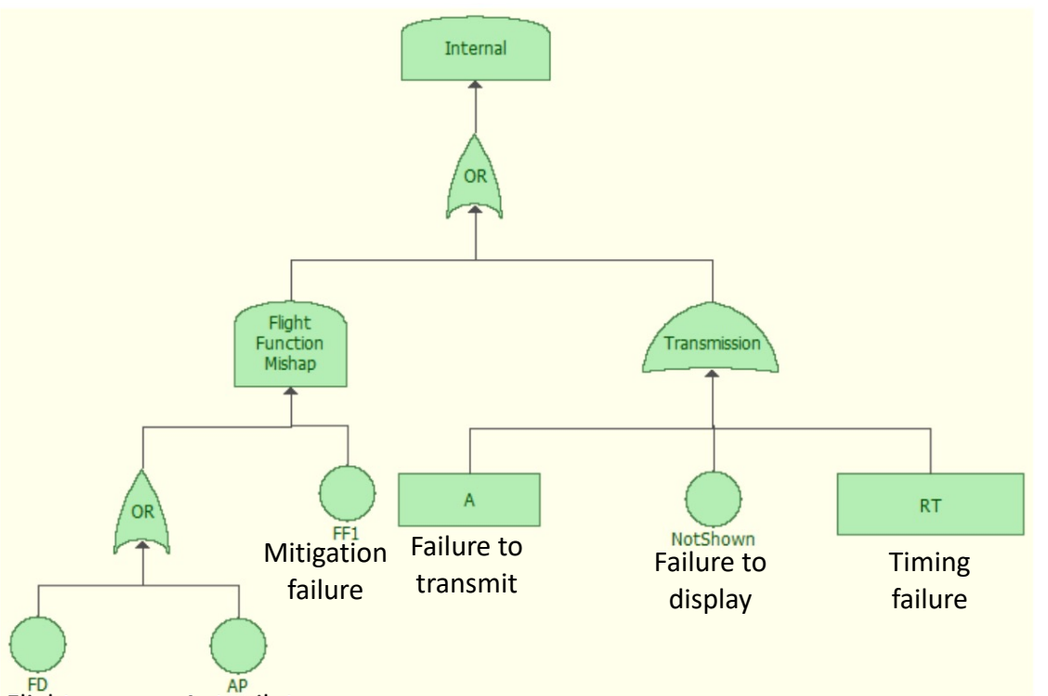
1. Identify components of the system and their functions and failure modes
   - How could each module/class/variable “fail”?
   - Failure to execute, incomplete execution, wrong timing, incorrect value
2. What are the effects of failure on other components and the system?
   - For each failure mode
3. What is the risk (probability) of occurrence and severity of each failure?
   - What would cause each failure? How likely is each cause?
   - Relative ranking
4. How likely is the failure to be detected by current/planned controls?
   - Detect causes or failure
   - Relative ranking, higher is **less likely** to be detected
5. Rank by the Risk Priority Number (RPN)
   - RPN = severity x occurrence x detection
6. Take action to reduce the severity, occurrence, or detection rankings of the highest RPN
7. Reevaluate

https://peer.asee.org/applying-fmea-to-software

Error Handling: FTA

Fault Tree Analysis: Identify top-level failure modes first, backtrack to list possible causes and chains of events, combine with logic AND, OR

- Top-down approach



[Fault tree analysis for safety/security verification in aviation software]

Error Handling: Summary

- Anticipate likely errors, try to avoid them in the first place
- Handle errors in context, preferably in the same place they were detected
  - May need to pass up to the caller instead
- Weigh the risk of not handling vs. cost of handling exceptions
  - Evaluate probability/severity of errors
- Handle errors systematically/consistently across the system
  - Have a planned strategy

Thank you!

Best of luck!