Lecture 13

ECE 1145: Software Construction and Evolution

Midterm Review

Announcements

- **Midterm Oct. 18 (take-home) – one week from today**
  - Open book, open notes, work individually
  - Access and submit via Canvas
  - 24 hour window
  - Lectures 1 – 9, project iterations 1 – 3 and code review
  - Midterm review on Wednesday Oct. 13
- Iteration 4 due **Oct. 24** - code quality improvements

Software Construction and Evolution

From the *Guide to the Software Engineering Body of Knowledge (SWEBOK),* 2004 edition:

> **Software construction** refers to the detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging.

“Growing” software does not mean we can’t have a plan or design

It means we are **agile** and prepared to change the plan (even dramatically) as we **learn from growing the software!**

Lecture 1

Software Quality Metrics

ISO 25010 (2011 – present):

Usability

Functional Suitability

Performance Efficiency

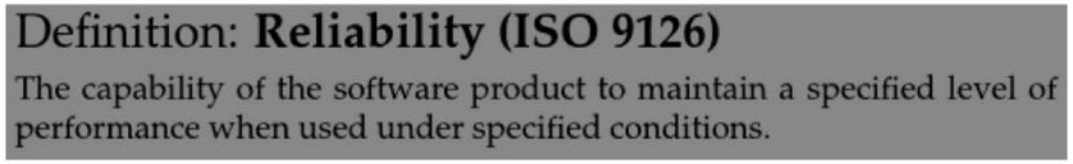
Portability

Reliability

Maintainability

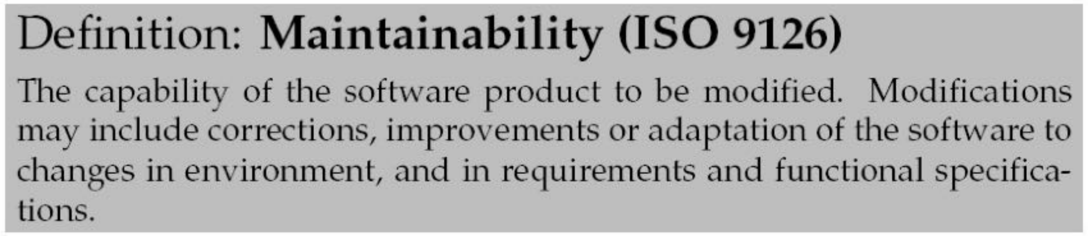
Security

Compatibility



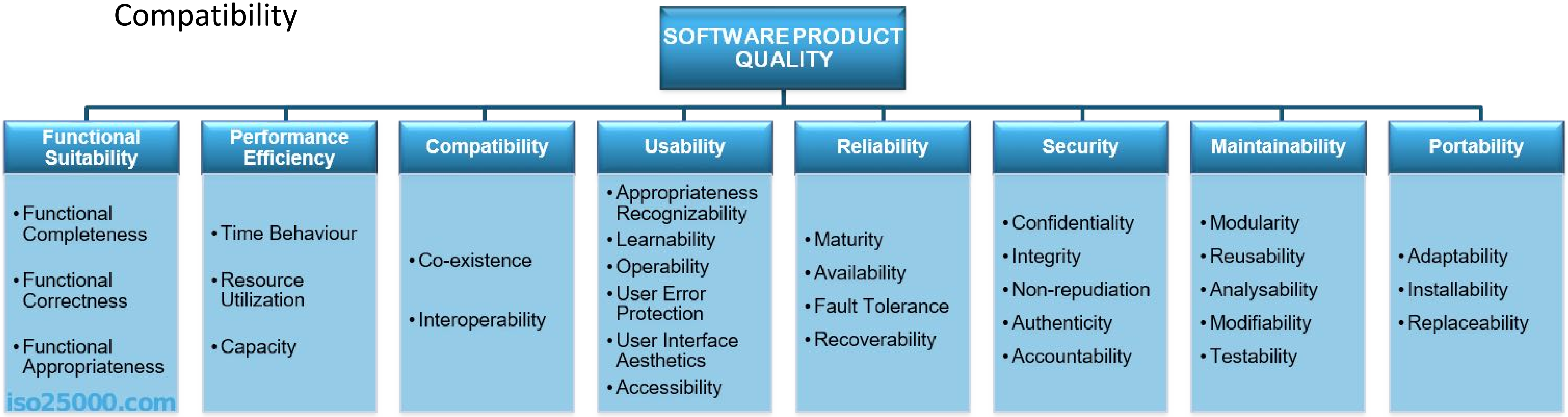
Definition: **Reliability (ISO 9126)**

The capability of the software product to maintain a specified level of performance when used under specified conditions.



Definition: **Maintainability (ISO 9126)**

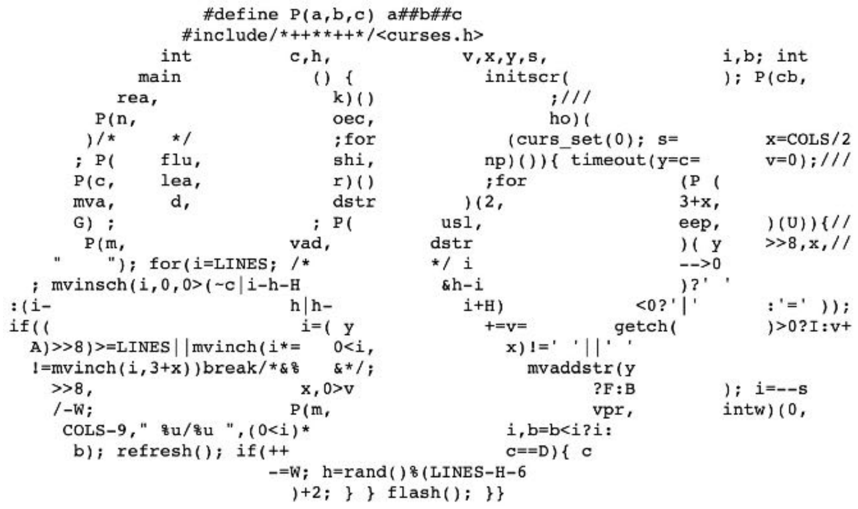
The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.



Maintainability

Customers / users of software likely don’t care if the code is readable, understandable, well documented… as long as it **works.**

→ **Reliability**



```
                  #define P(a,b,c) a##b##c
                #include/*++***++*/<curses.h>
              int         c,h,            v,x,y,s,                i,b; int
            main            () {            initscr(              ); P(cb,
          rea,                k)()                ;///
        P(n,                  oec,                ho)(
       )/*     */             ;for            (curs_set(0); s=        x=COLS/2
      ; P(    flu,            shi,          np)()){ timeout(y=c=      v=0);///
      P(c,    lea,            r)()          ;for              (P (
      mva,     d,             dstr        )(2,                3+x,
      G) ;                  ; P(        usl,                  eep,    )(U)){//
       P(m,               vad,         dstr                   )( y    >>8,x,//
    "    "); for(i=LINES; /*           */ i                   -->0
  ; mvinsch(i,0,0>(~c|i-h-H             &h-i                  )?' '
:(i-                      h|h-            i+H)            <0?'|'      :'=' ));
if((                       i=( y            +=v=        getch(        )>0?I:v+
  A)>>8)>=LINES||mvinch(i*=   0<i,            x)!=' '||' '
  !=mvinch(i,3+x))break/*&%   &*/;              mvaddstr(y
    >>8,                   x,0>v                      ?F:B        ); i=--s
    /-W;                  P(m,                        vpr,        intw)(0,
     COLS-9," %u/%u ",(0<i)*                  i,b=b<i?i:
      b); refresh(); if(++                    c==D){ c
                        -=W; h=rand()%(LINES-H-6
                          )+2; } } flash(); }}
```

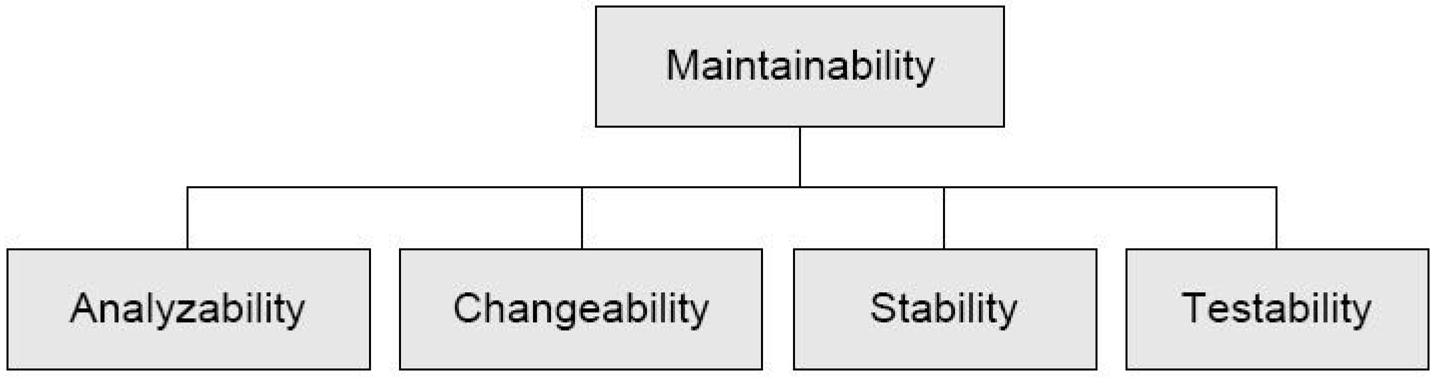
Code quality is usually most important to **developers** to ensure **maintainability.**

→ Although, good code also tends to have fewer bugs, because it is more **testable** (and the bugs are more fixable, due to **analyzability** and **modifiability**)

Maintainability

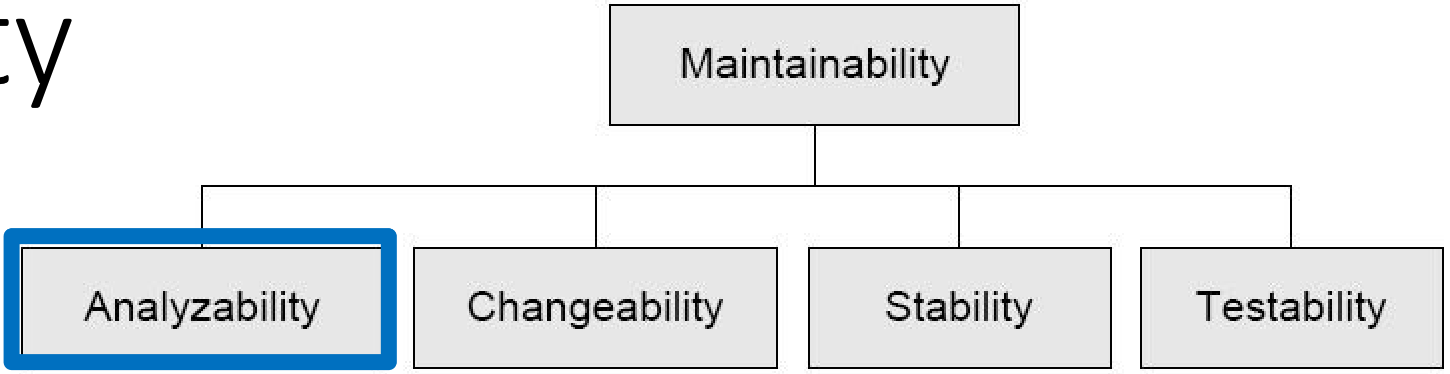
Definition: **Maintainability (ISO 9126)**

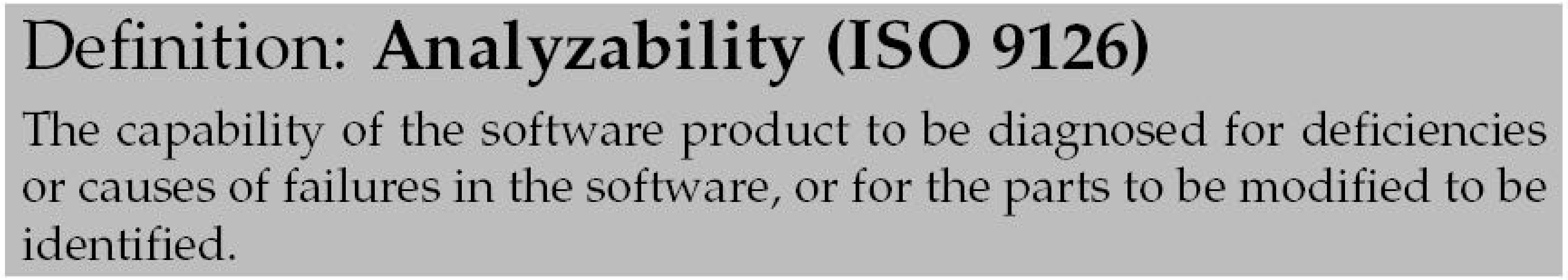
The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.



(Modifiability)

Maintainability



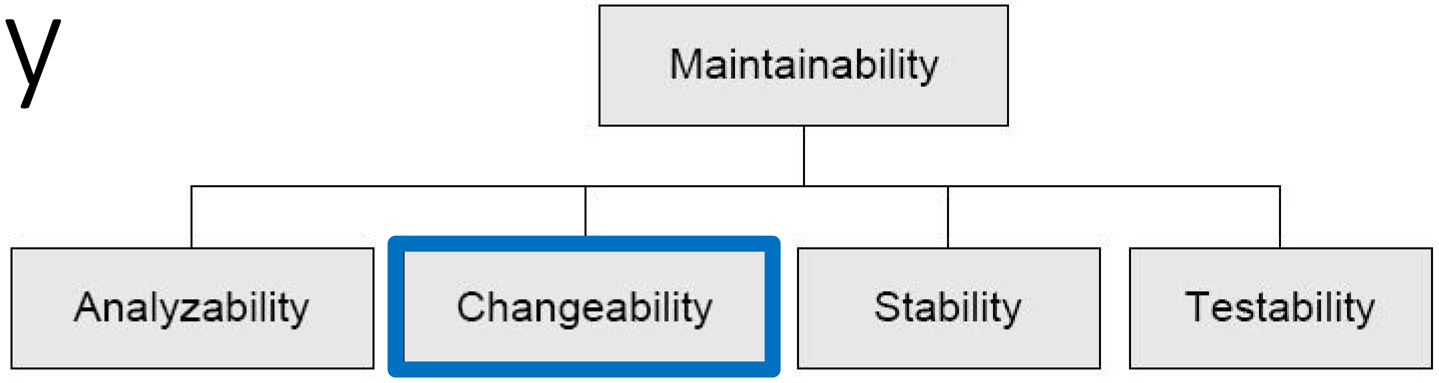

Definition: **Analyzability (ISO 9126)**

The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.

Can we **understand** the code?

- Indentation
- Naming conventions for classes/methods
- Useful comments
- Descriptive variable names
- Training, e.g., to recognize **design patterns**

Maintainability



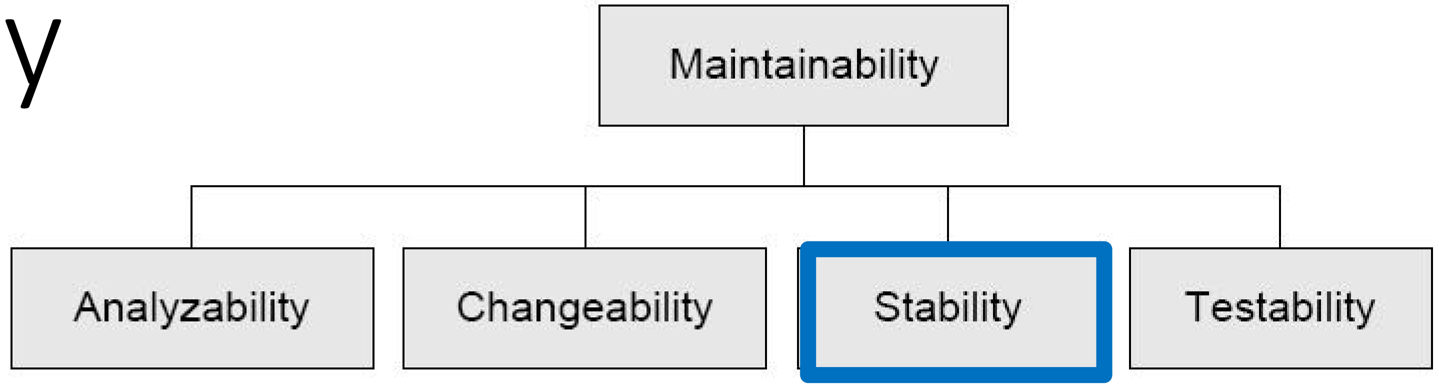
Definition: Changeability (ISO 9126)

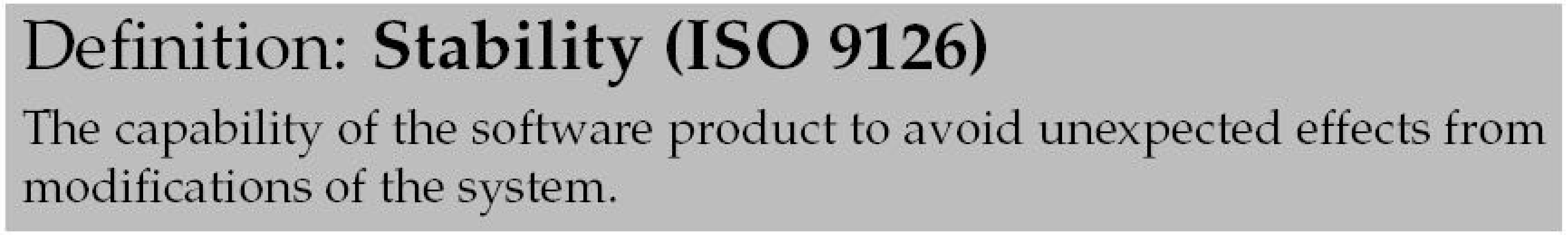
The capability of the software product to enable a specified modification to be implemented.

What is the **cost** to modify the code?

- Cost: money, time, personnel, etc.
- Example: Use named variables instead of hardcoded “magic numbers” throughout the code
- Will discuss more with design patterns and framework theory

Maintainability



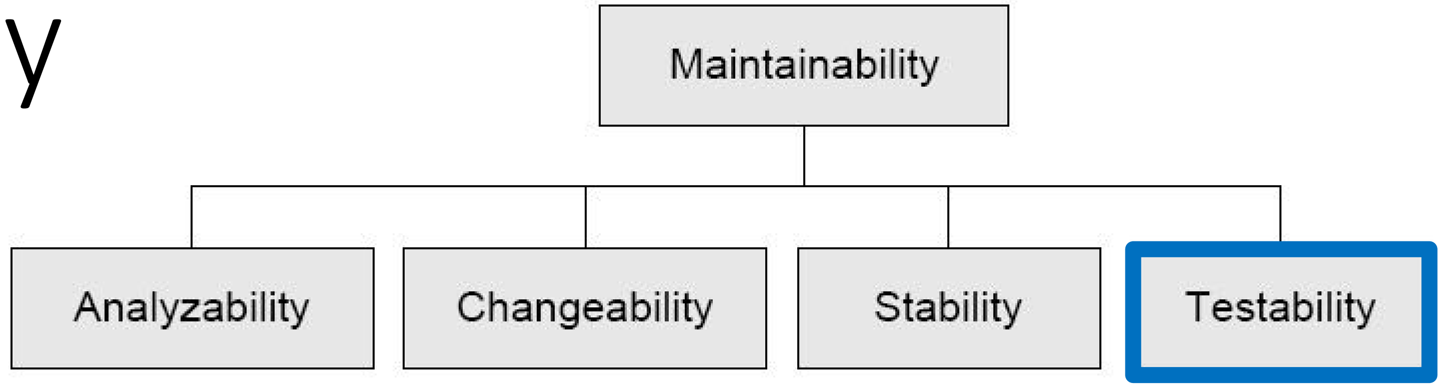

Definition: **Stability (ISO 9126)**

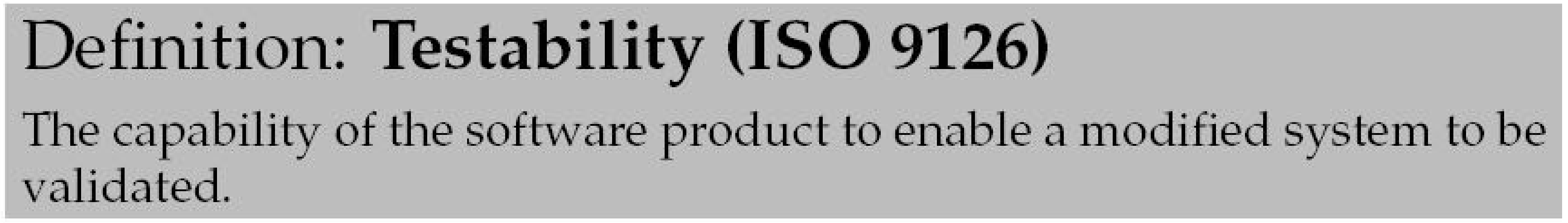
The capability of the software product to avoid unexpected effects from modifications of the system.

What are **potential (negative) effects** of modifying the code?

- Will discuss more with design patterns, integration testing, and compositional design

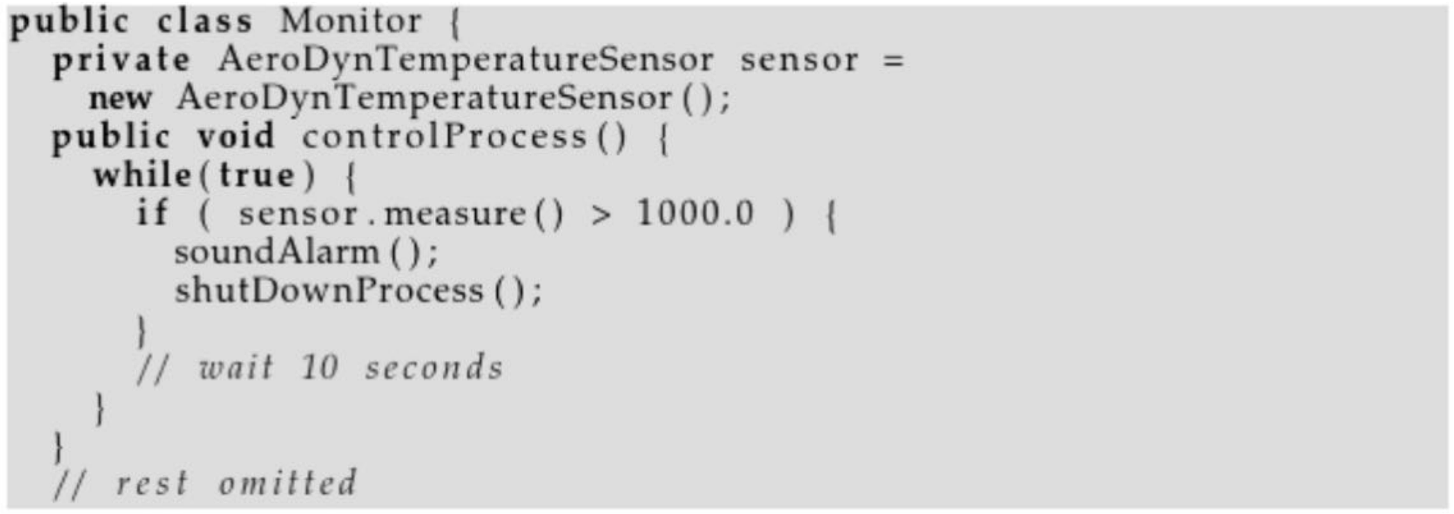
Maintainability



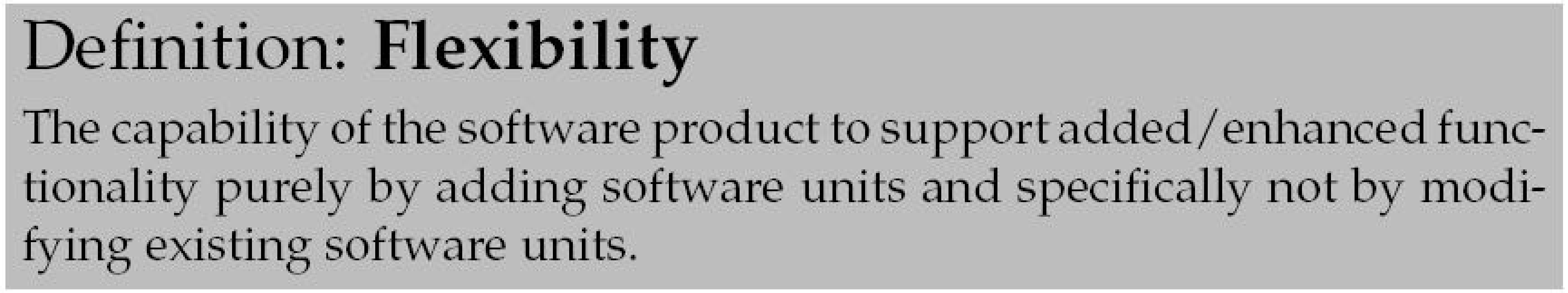

Definition: Testability (ISO 9126)

The capability of the software product to enable a modified system to be validated.

Can we **verify** the code? (with tests)


```
public class Monitor {
  private AeroDynTemperatureSensor sensor =
    new AeroDynTemperatureSensor();
  public void controlProcess() {
    while(true) {
      if ( sensor.measure() > 1000.0 ) {
        soundAlarm();
        shutDownProcess();
      }
      // wait 10 seconds
    }
  }
  // rest omitted
```

Flexibility

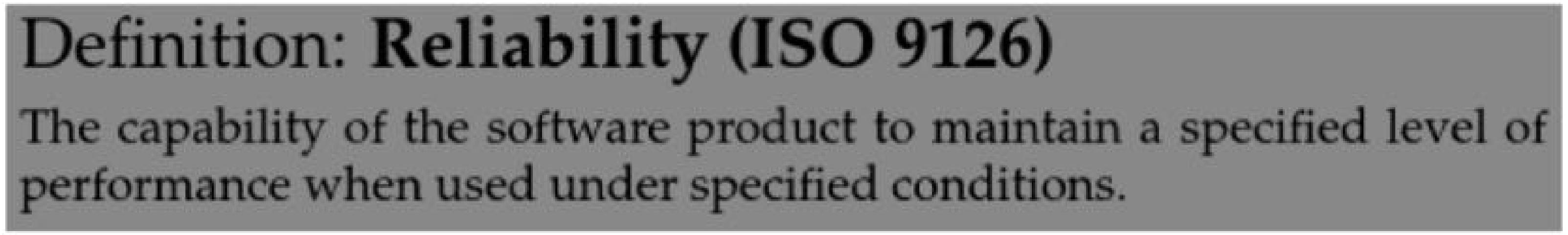


Definition: **Flexibility**

The capability of the software product to support added/enhanced functionality purely by adding software units and specifically not by modifying existing software units.

- A special case of **changeability**
- Also relates to **stability**
- **Change by addition, not by modification**

Reliability



Definition: **Reliability (ISO 9126)**

The capability of the software product to maintain a specified level of performance when used under specified conditions.

How do we build reliable software?

- Utilize programming language constructs (e.g., variable scope) and established coding techniques (e.g., patterns)
- **Review** – ensure understandable code and documentation
- **Test –** execute software in order to find situations where it does not perform its required function (try to break it!)
- **Test-Driven Development** - quickly produce reliable and maintainable software through a well-structured programming process

Lecture 3

Testing

Definition: Test case

A test case is a definition of input values and expected output values for a unit under test.

Definition: Test suite

A test suite is a set of test cases.

Unit testing tools support executing large test suites, reporting, and finding test cases that fail

Example: JUnit, Hamcrest

With Hamcrest:

```
assertThat(date.dayOfWeek(),
    is(Date.Weekday.SATURDAY));
```

```
import org.junit.*;
import static org.junit.Assert.*;

/** Testing dayOfWeek using the JUnit 4.x framework.
*/
public class TestDayOfWeek {

  /**
   * Test that December 25th 2010 is Saturday
   */
  @Test
  public void shouldGiveSaturdayFor25Dec2010() {
    Date date = new Date( 2010, 12, 25);
    assertEquals( "Dec 25th 2010 is Saturday",
                  Date.Weekday.SATURDAY, date.dayOfWeek() );
  }
}
```

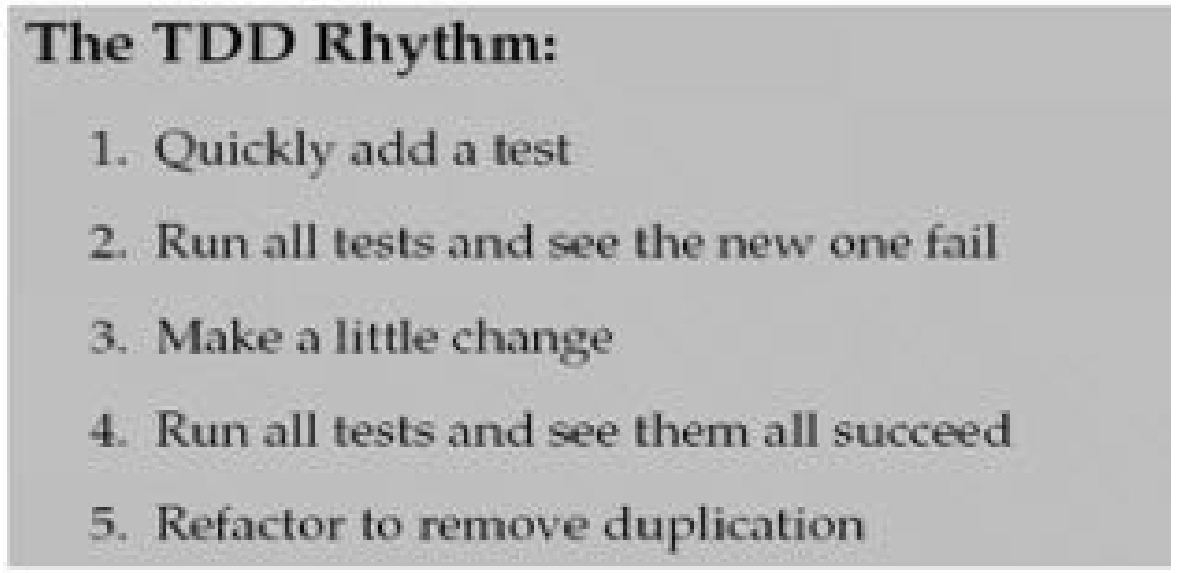
← Annotated method defines a test case, all methods marked constitute a test suite

^ Use assert methods for comparing output with expected output

Test-Driven Development

Tests can be considered a **specification** of desired behavior.

Test-Driven Development is a systematic programming technique focused on using tests to ensure continued reliability, especially with changes to the code.



The TDD Rhythm:

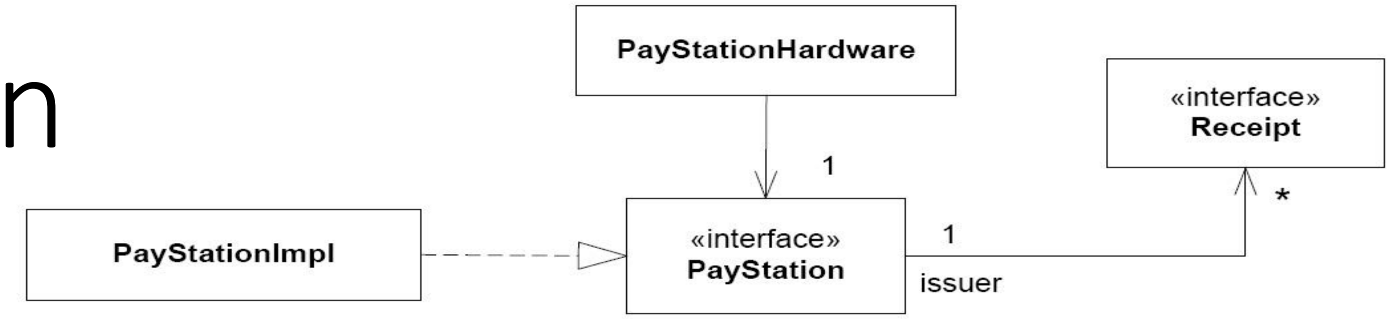
1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

(6. All tests pass again after refactoring!)

Case: PayStation

Welcome to *PayStation Ltd.*

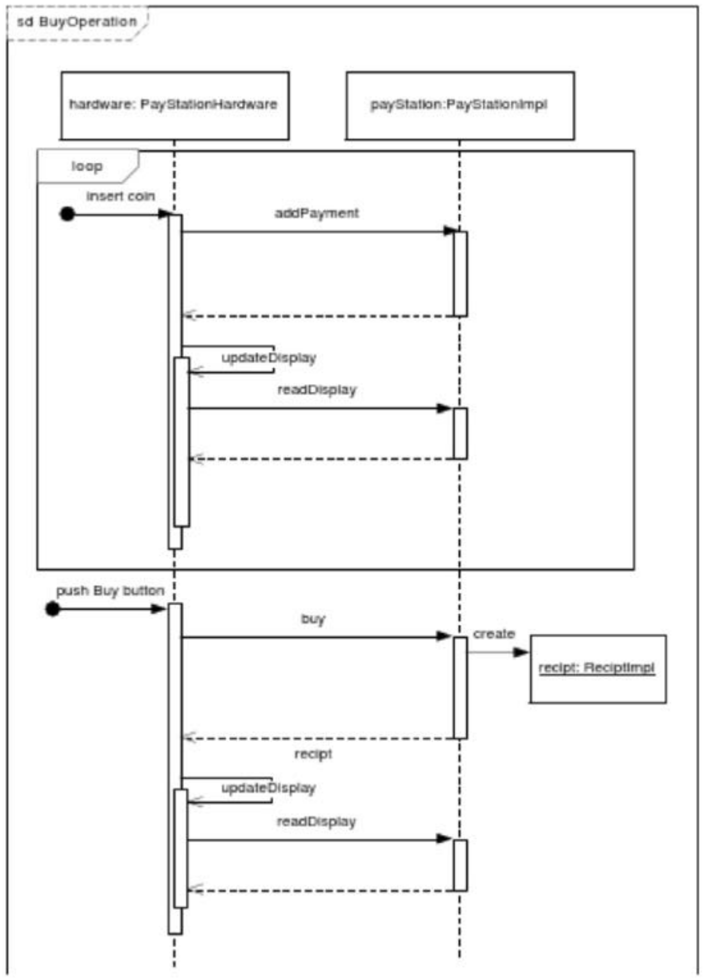
Customer: AlphaTown



Requirements

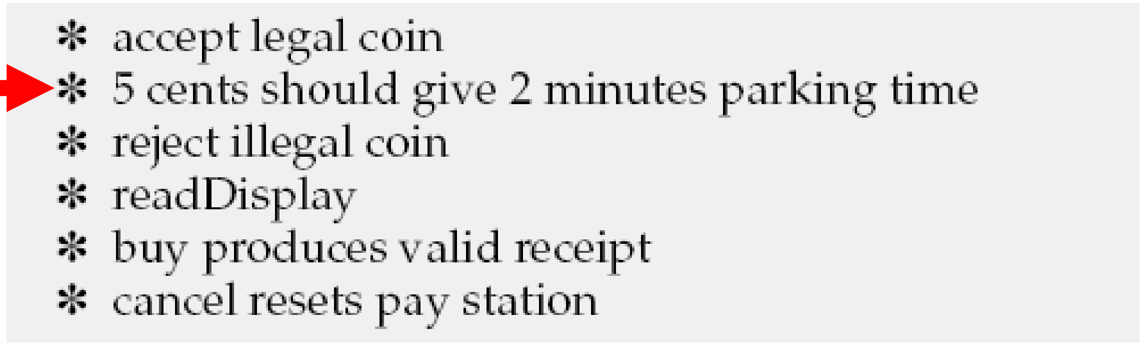
- accept coins for payment
  - 5, 10, 25 c.
- show time bought on display
- print parking time receipts
- 2 minutes costs 5 c.
- handle buy and cancel

Test list!

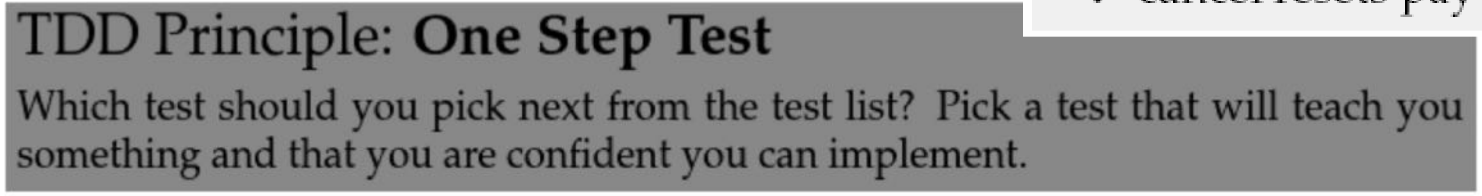


- ✻ accept legal coin
- ✻ 5 cents should give 2 minutes parking time
- ✻ reject illegal coin
- ✻ readDisplay
- ✻ buy produces valid receipt
- ✻ cancel resets pay station

Case: PayStation

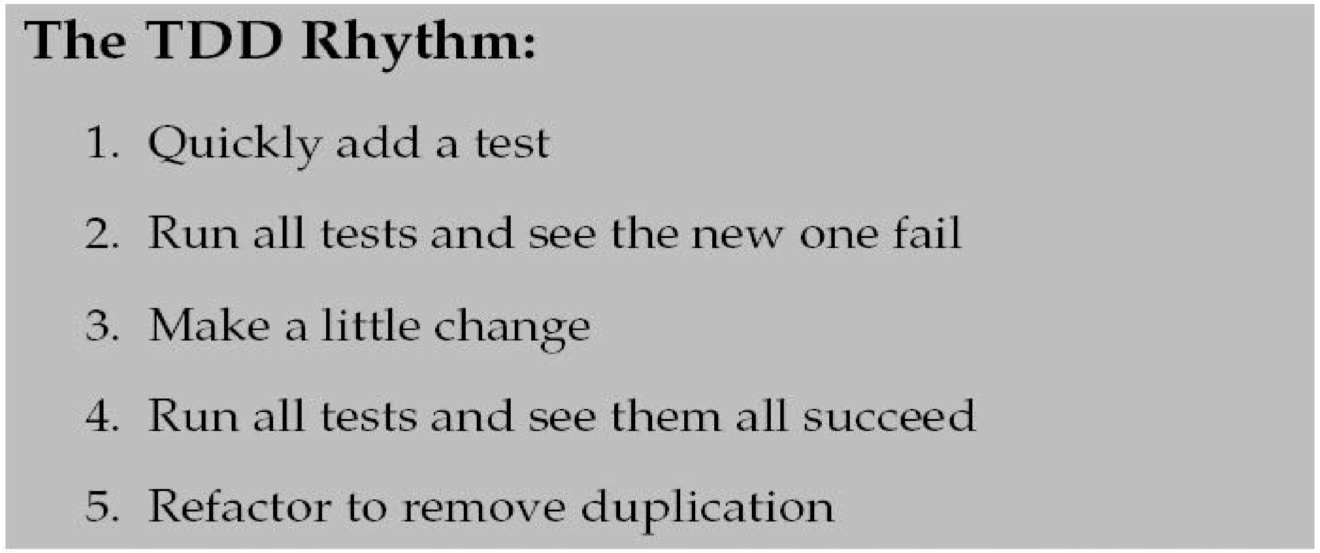


- ✻ accept legal coin
- ✻ 5 cents should give 2 minutes parking time
- ✻ reject illegal coin
- ✻ readDisplay
- ✻ buy produces valid receipt
- ✻ cancel resets pay station



TDD Principle: **One Step Test**

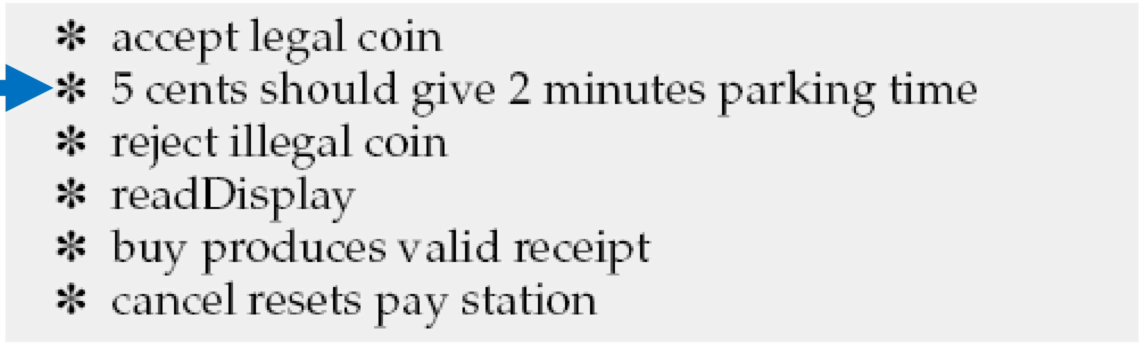
Which test should you pick next from the test list? Pick a test that will teach you something and that you are confident you can implement.



The TDD Rhythm:

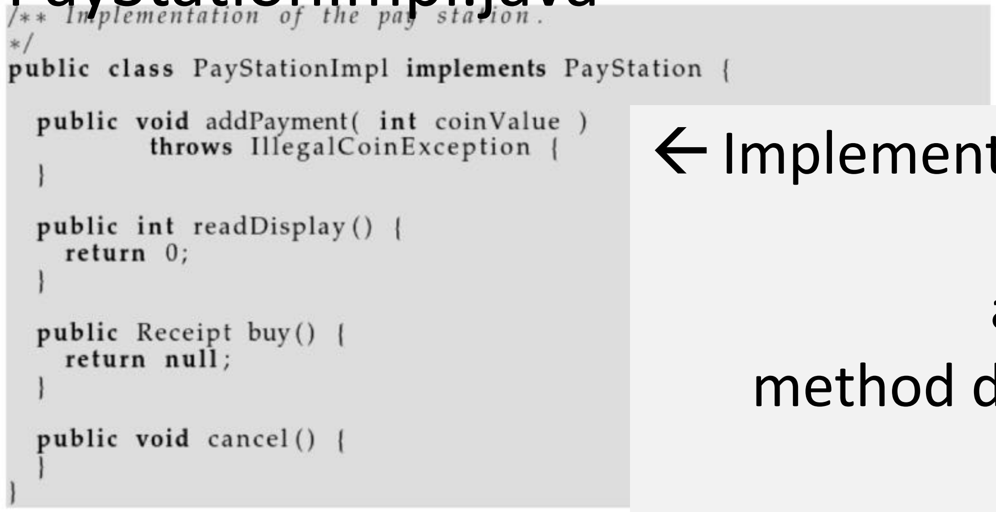
1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

Case: PayStation



- ✻ accept legal coin
- ✻ 5 cents should give 2 minutes parking time
- ✻ reject illegal coin
- ✻ readDisplay
- ✻ buy produces valid receipt
- ✻ cancel resets pay station

PayStationImpl.java

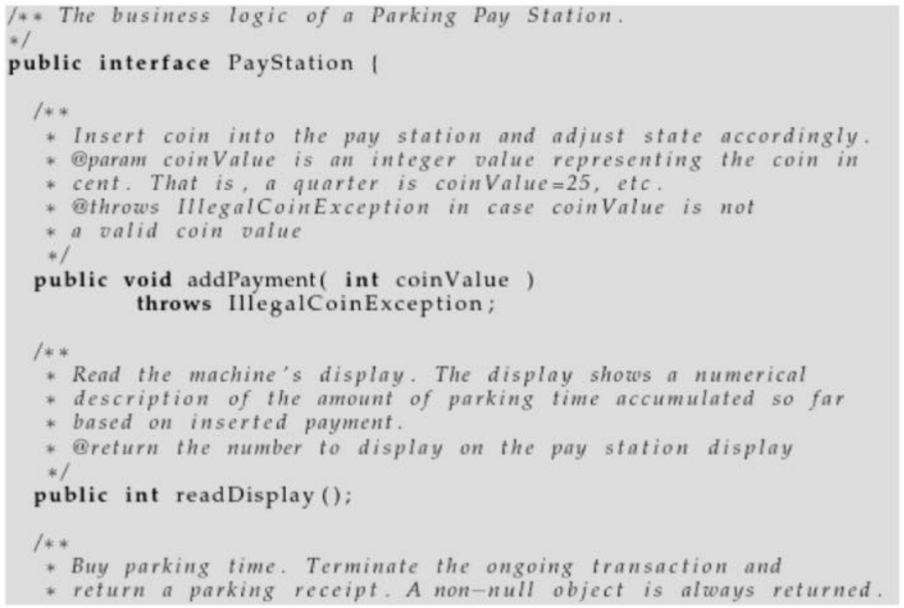


```
/** Implementation of the pay station.
*/
public class PayStationImpl implements PayStation {

  public void addPayment( int coinValue )
          throws IllegalCoinException {
  }

  public int readDisplay() {
    return 0;
  }

  public Receipt buy() {
    return null;
  }

  public void cancel() {
  }
}
```

← Implementation here,

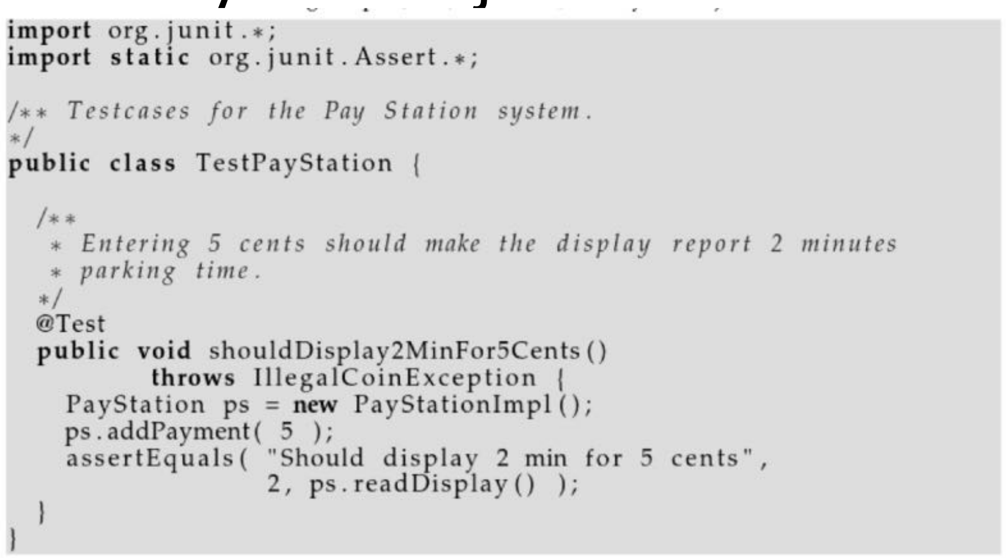
according to method definitions in interface →

PayStation.java



```
/** The business logic of a Parking Pay Station.
*/
public interface PayStation {

  /**
   * Insert coin into the pay station and adjust state accordingly.
   * @param coinValue is an integer value representing the coin in
   * cent. That is, a quarter is coinValue=25, etc.
   * @throws IllegalCoinException in case coinValue is not
   * a valid coin value
   */
  public void addPayment( int coinValue )
          throws IllegalCoinException;

  /**
   * Read the machine's display. The display shows a numerical
   * description of the amount of parking time accumulated so far
   * based on inserted payment.
   * @return the number to display on the pay station display
   */
  public int readDisplay();

  /**
   * Buy parking time. Terminate the ongoing transaction and
   * return a parking receipt. A non-null object is always returned.
```

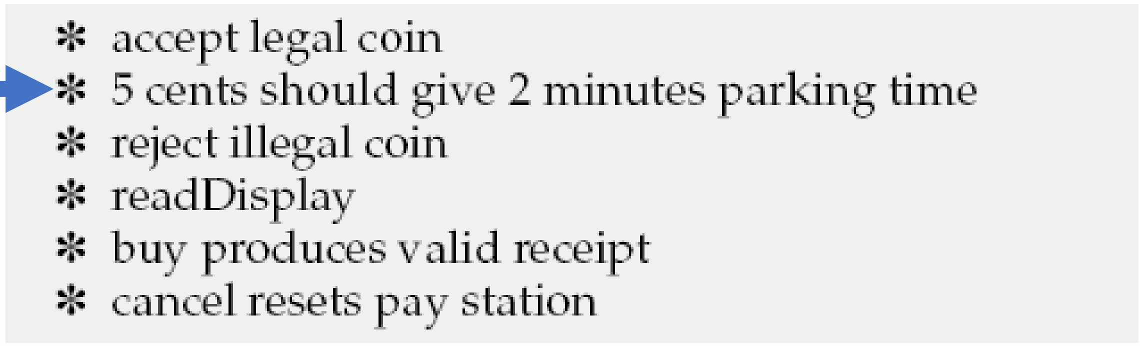
TestPayStation.java



```
import org.junit.*;
import static org.junit.Assert.*;

/** Testcases for the Pay Station system.
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display report 2 minutes
   * parking time.
   */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }
}
```

← Tests go here

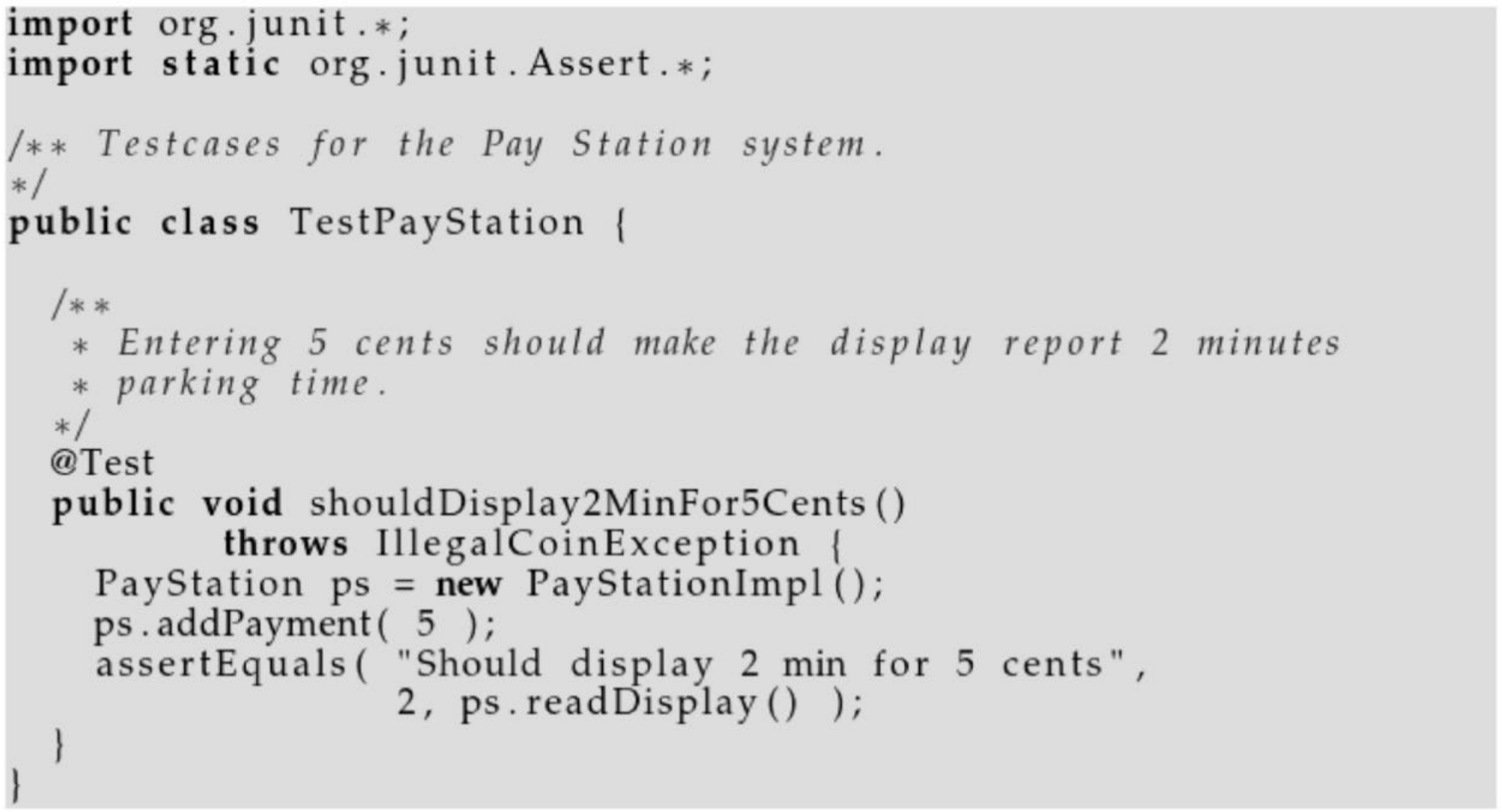
Case: PayStation



- ✻ accept legal coin
- ✻ 5 cents should give 2 minutes parking time
- ✻ reject illegal coin
- ✻ readDisplay
- ✻ buy produces valid receipt
- ✻ cancel resets pay station

Display 2 minutes if input value is 5 cents (addPayment)

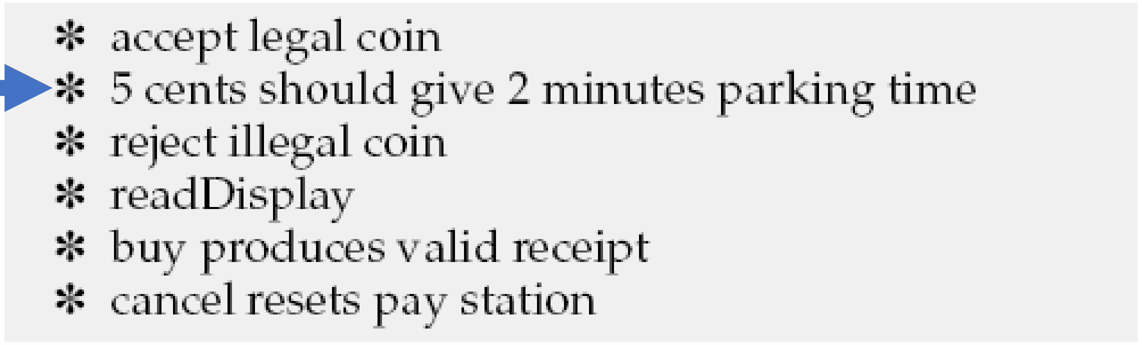
TestPayStation.java



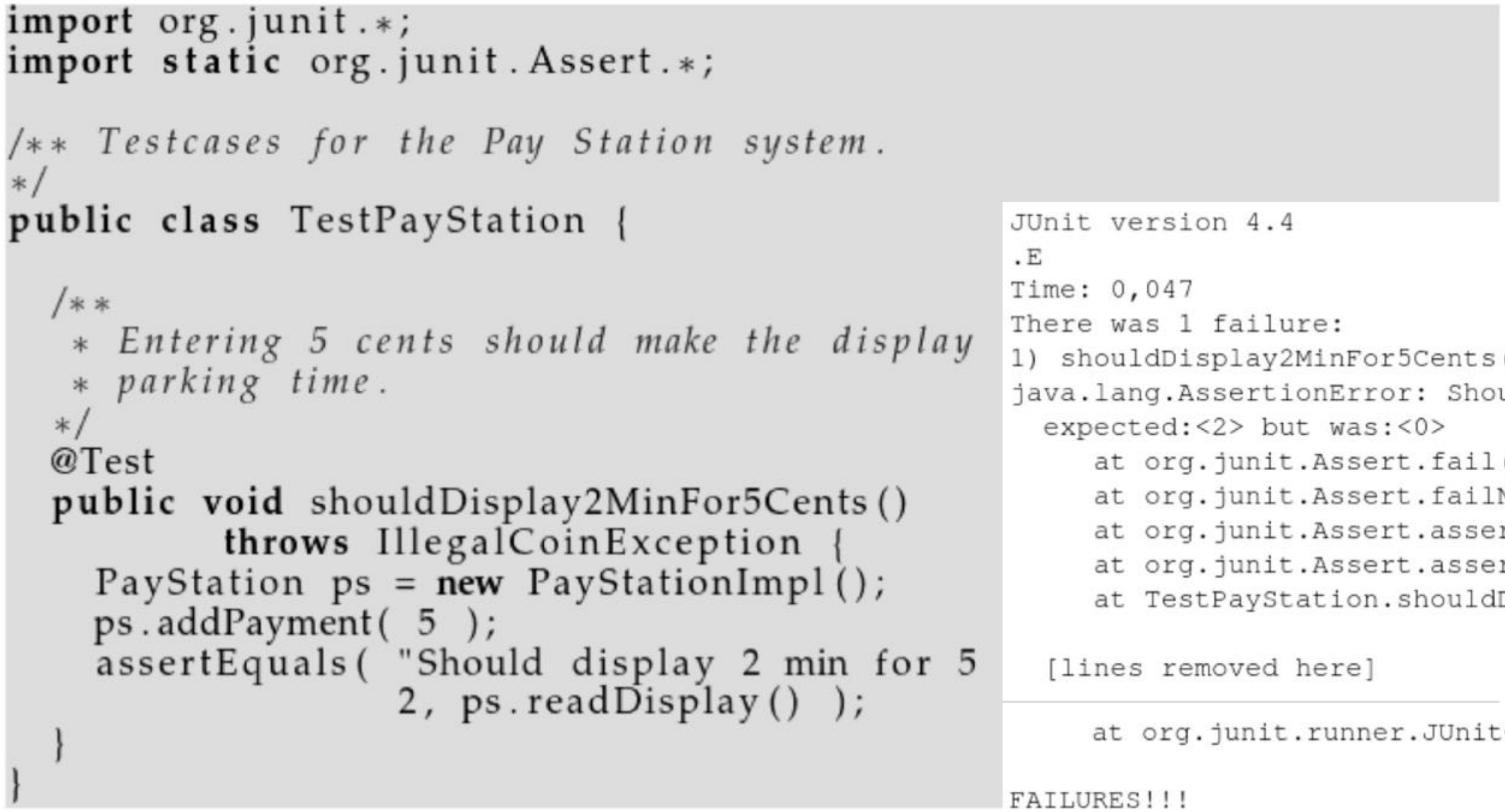
```
import org.junit.*;
import static org.junit.Assert.*;

/** Testcases for the Pay Station system.
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display report 2 minutes
   * parking time.
  */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }
}
```

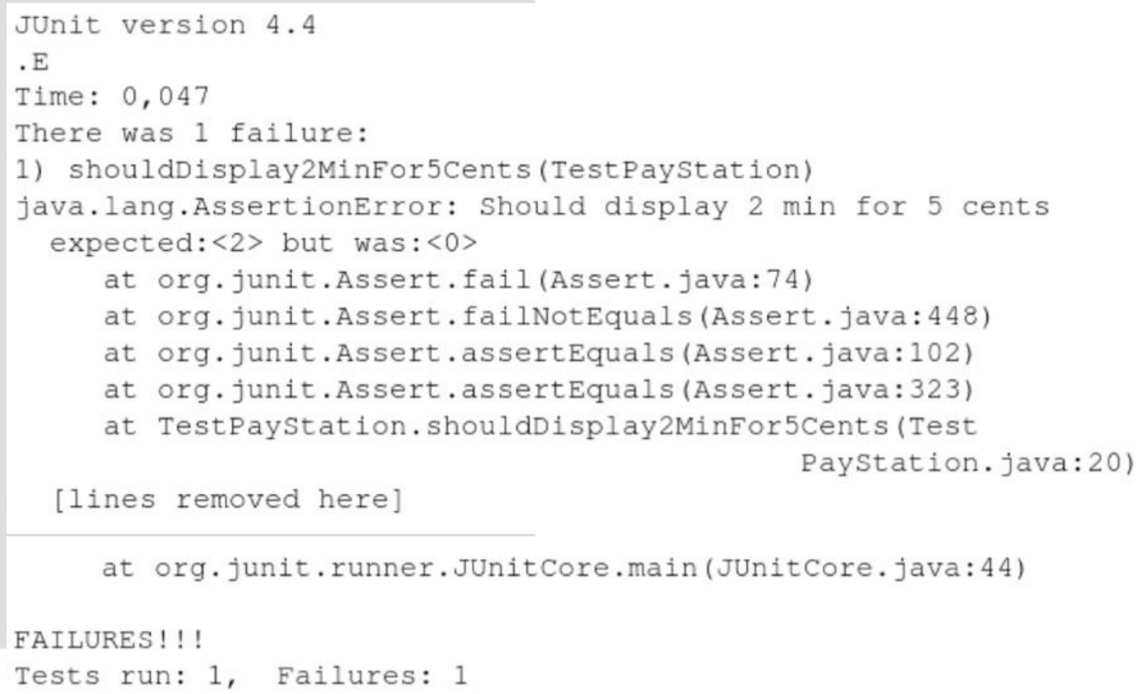
1. Add a test

Case: PayStation

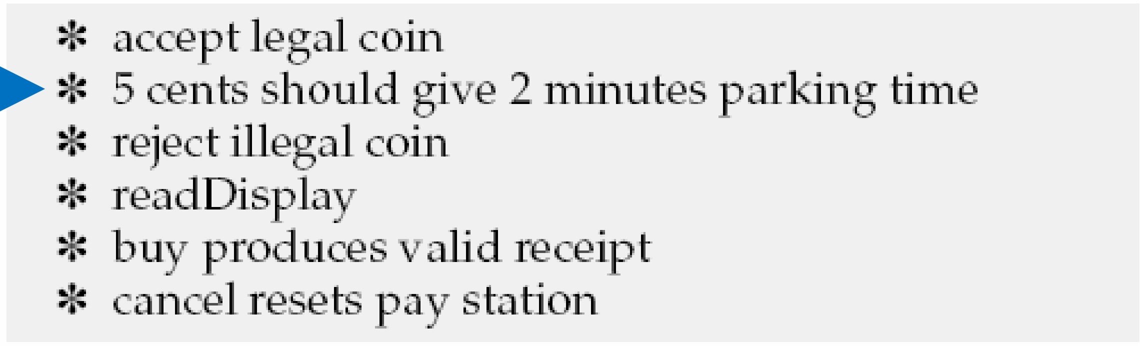

- accept legal coin
- 5 cents should give 2 minutes parking time
- reject illegal coin
- readDisplay
- buy produces valid receipt
- cancel resets pay station

2. See the test fail (as expected)!


```
import org.junit.*;
import static org.junit.Assert.*;

/** Testcases for the Pay Station system.
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display
   * parking time.
  */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5
                  2, ps.readDisplay() );
  }
}
```


```
JUnit version 4.4
.E
Time: 0,047
There was 1 failure:
1) shouldDisplay2MinFor5Cents(TestPayStation)
java.lang.AssertionError: Should display 2 min for 5 cents
  expected:<2> but was:<0>
     at org.junit.Assert.fail(Assert.java:74)
     at org.junit.Assert.failNotEquals(Assert.java:448)
     at org.junit.Assert.assertEquals(Assert.java:102)
     at org.junit.Assert.assertEquals(Assert.java:323)
     at TestPayStation.shouldDisplay2MinFor5Cents(Test
                                             PayStation.java:20)
  [lines removed here]

     at org.junit.runner.JUnitCore.main(JUnitCore.java:44)

FAILURES!!!
Tests run: 1,  Failures: 1
```

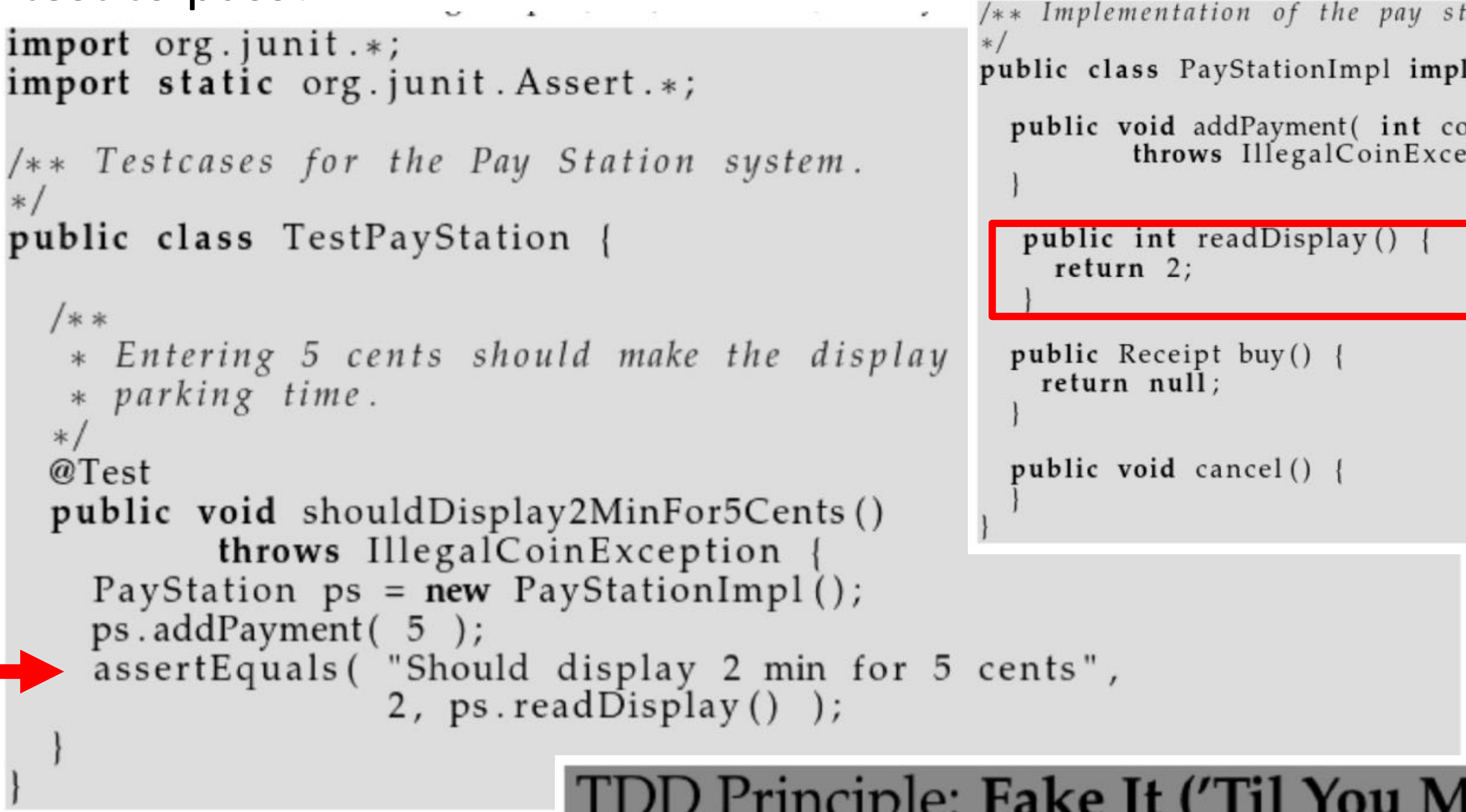
Case: PayStation



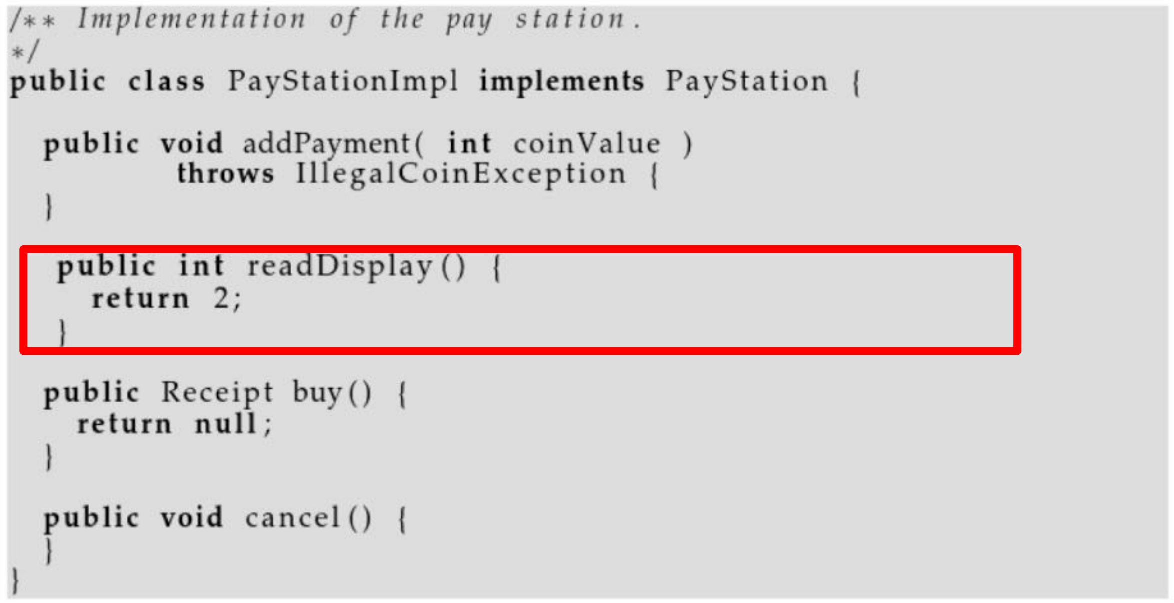
- accept legal coin
- 5 cents should give 2 minutes parking time
- reject illegal coin
- readDisplay
- buy produces valid receipt
- cancel resets pay station

3. Make a little change

What is the simplest possible PayStation class implementation that will cause the test to pass?



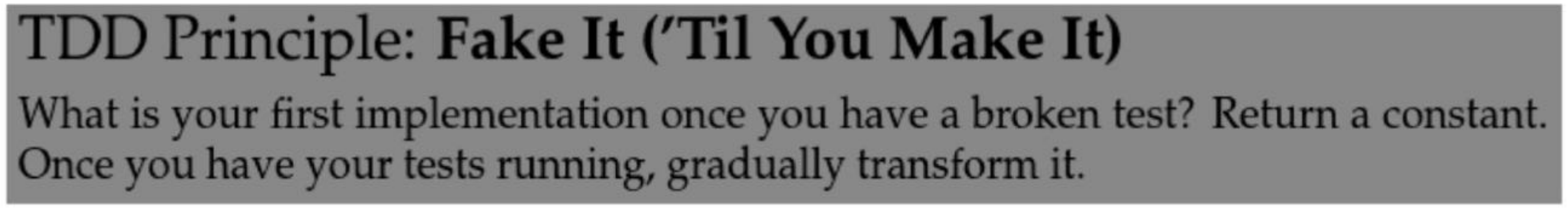
```java
import org.junit.*;
import static org.junit.Assert.*;

/** Testcases for the Pay Station system.
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display
   * parking time.
  */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }
}
```



```java
/** Implementation of the pay station.
*/
public class PayStationImpl implements PayStation {

  public void addPayment( int coinValue )
          throws IllegalCoinException {
  }

  public int readDisplay() {
    return 2;
  }

  public Receipt buy() {
    return null;
  }

  public void cancel() {
  }
}
```

PayStationImpl

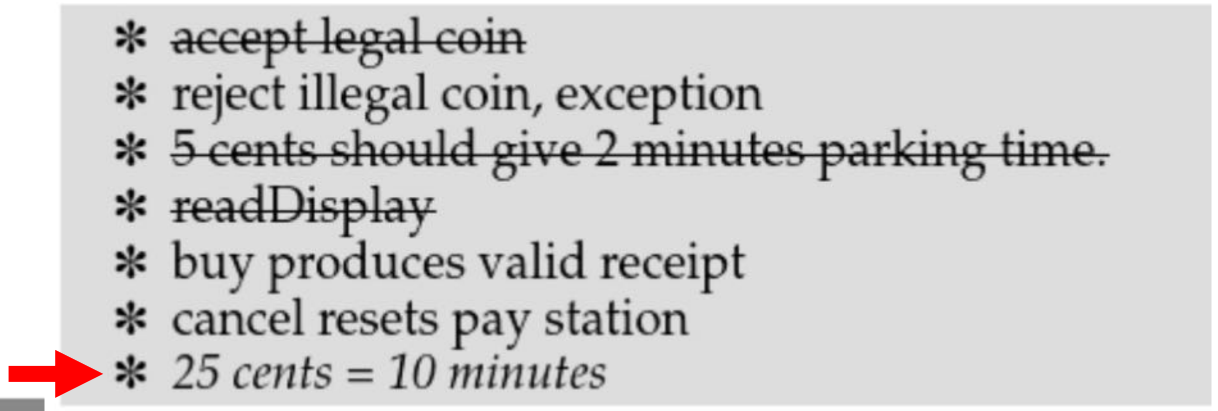
→ 4. See Tests Pass



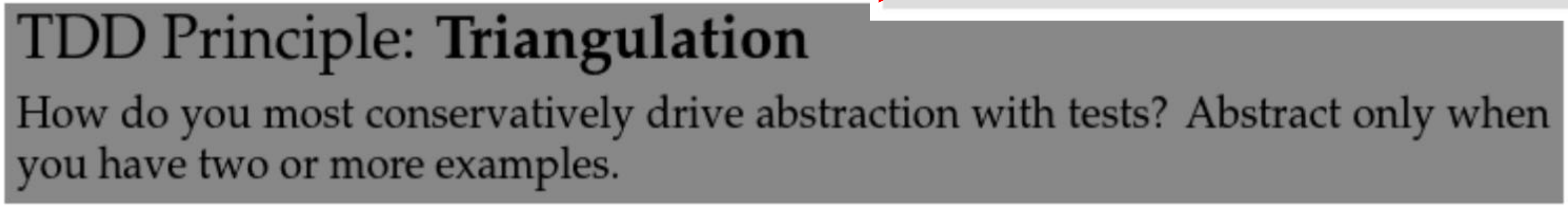
TDD Principle: Fake It ('Til You Make It)

What is your first implementation once you have a broken test? Return a constant. Once you have your tests running, gradually transform it.

Case: PayStation



- ✻ ~~accept legal coin~~
- ✻ reject illegal coin, exception
- ✻ ~~5 cents should give 2 minutes parking time.~~
- ✻ ~~readDisplay~~
- ✻ buy produces valid receipt
- ✻ cancel resets pay station
- → ✻ *25 cents = 10 minutes*



TDD Principle: Triangulation

How do you most conservatively drive abstraction with tests? Abstract only when you have two or more examples.

You should have two, three, or more examples in order to generalize and drive an abstraction like an algorithm or class into existence

→ Wait to implement the parking time calculation until it is necessary

→ Add a test case to the test list as a reminder

Case: PayStation

1. Add a test

- ✻ ~~accept legal coin~~
- ✻ reject illegal coin, exception
- ✻ ~~5 cents should give 2 minutes parking time.~~
- ✻ ~~readDisplay~~
- ✻ buy produces valid receipt
- ✻ cancel resets pay station
- → ✻ *25 cents = 10 minutes*

```
@Test
public void shouldDisplay2MinFor5Cents()
        throws IllegalCoinException {
  PayStation ps = new PayStationImpl();
  ps.addPayment( 5 );
  assertEquals( "Should display 2 min for 5 cents",
                2, ps.readDisplay() );

  ps.addPayment( 25 );
  assertEquals( "Should display 10 min for 25 cents",
                10, ps.readDisplay() );
}
```

✗

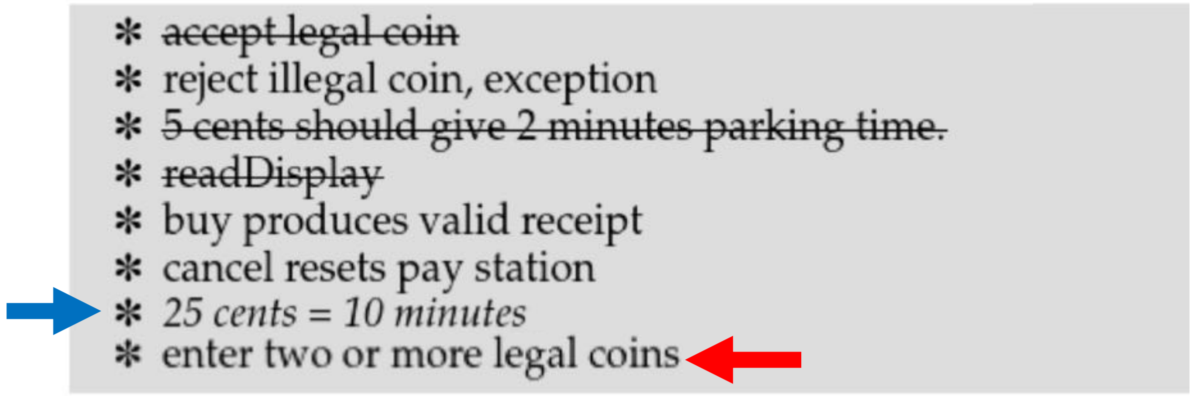
```
@Test
public void shouldDisplay10MinFor25Cents()
        throws IllegalCoinException {
  PayStation ps = new PayStationImpl();
  ps.addPayment( 25 );
  assertEquals( "Should display 10 min for 25 cents",
                10, ps.readDisplay() );
}
```

✓

TDD Principle: Isolated Test

How should the running of tests affect one another? Not at all.

Case: PayStation



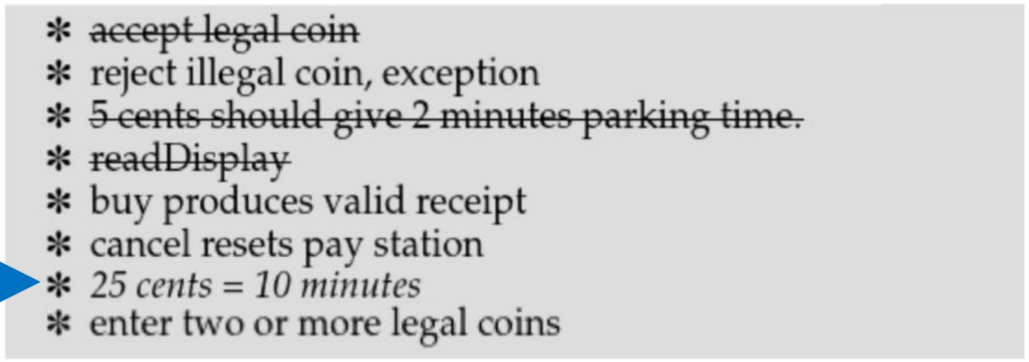
- ✻ ~~accept legal coin~~
- ✻ reject illegal coin, exception
- ✻ ~~5 cents should give 2 minutes parking time.~~
- ✻ ~~readDisplay~~
- ✻ buy produces valid receipt
- ✻ cancel resets pay station
- ✻ *25 cents = 10 minutes*
- ✻ enter two or more legal coins

Key point: Production code is driven into existence by tests
In the extreme, you do not enter a single character into production code unless there is a test case that demands it.

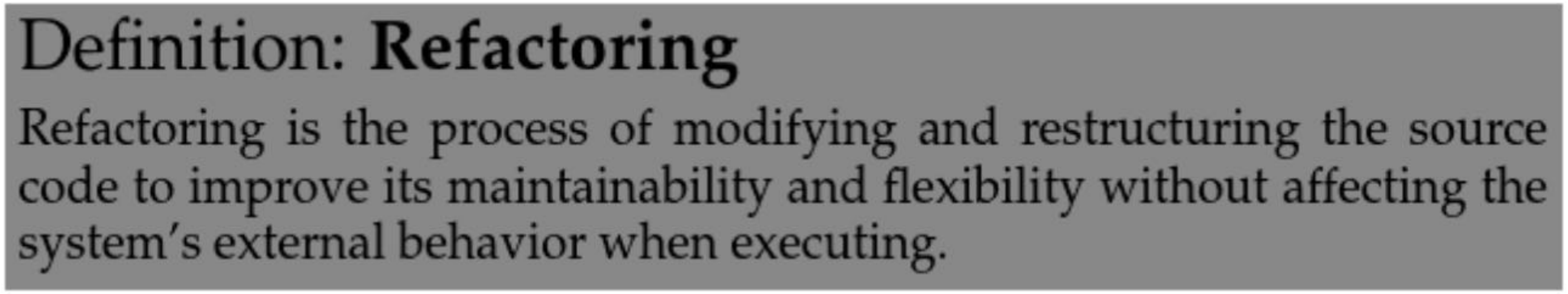
```
@Test
public void shouldDisplay10MinFor25Cents()
        throws IllegalCoinException {
  PayStation ps = new PayStationImpl();
  ps.addPayment( 25 );
  assertEquals( "Should display 10 min for 25 cents",
                10, ps.readDisplay() );
}
```

```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  public void addPayment( int coinValue )
          throws IllegalCoinException {
    insertedSoFar = coinValue;
  }
  public int readDisplay() {
    return insertedSoFar / 5 * 2;
  }
```

Case: PayStation

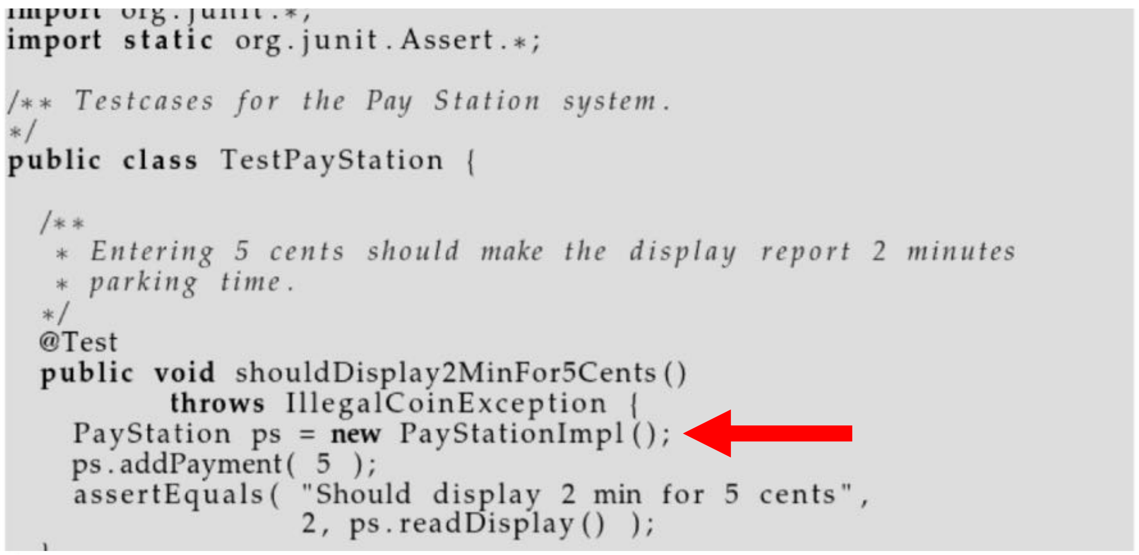


- ~~accept legal coin~~
- reject illegal coin, exception
- ~~5 cents should give 2 minutes parking time.~~
- ~~readDisplay~~
- buy produces valid receipt
- cancel resets pay station
- *25 cents = 10 minutes*
- enter two or more legal coins

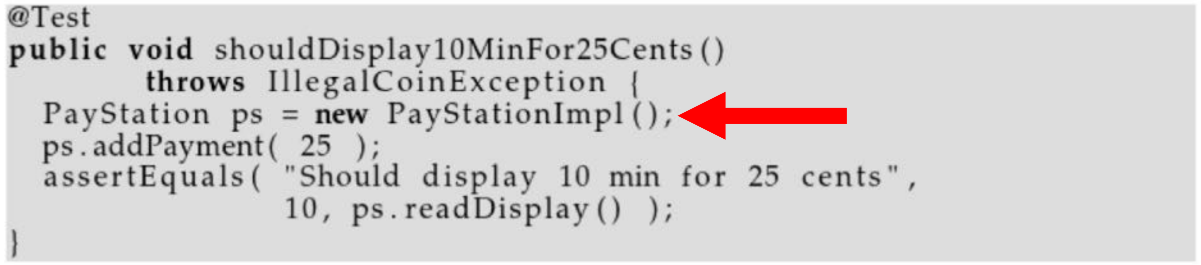


Definition: Refactoring

Refactoring is the process of modifying and restructuring the source code to improve its maintainability and flexibility without affecting the system's external behavior when executing.



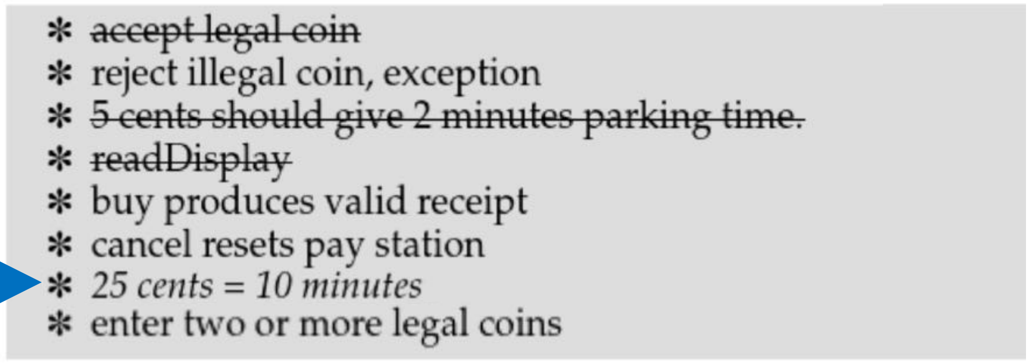
```
import org.junit.*;
import static org.junit.Assert.*;

/** Testcases for the Pay Station system.
*/
public class TestPayStation {

  /**
   * Entering 5 cents should make the display report 2 minutes
   * parking time.
  */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
```



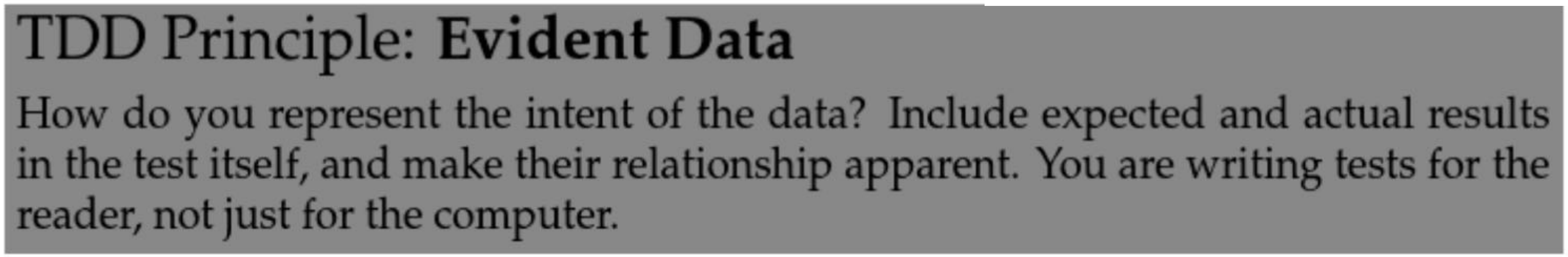
```
@Test
public void shouldDisplay10MinFor25Cents()
        throws IllegalCoinException {
  PayStation ps = new PayStationImpl();
  ps.addPayment( 25 );
  assertEquals( "Should display 10 min for 25 cents",
                10, ps.readDisplay() );
}
```

5. Refactor to remove duplication (also applies to test code)

Case: PayStation

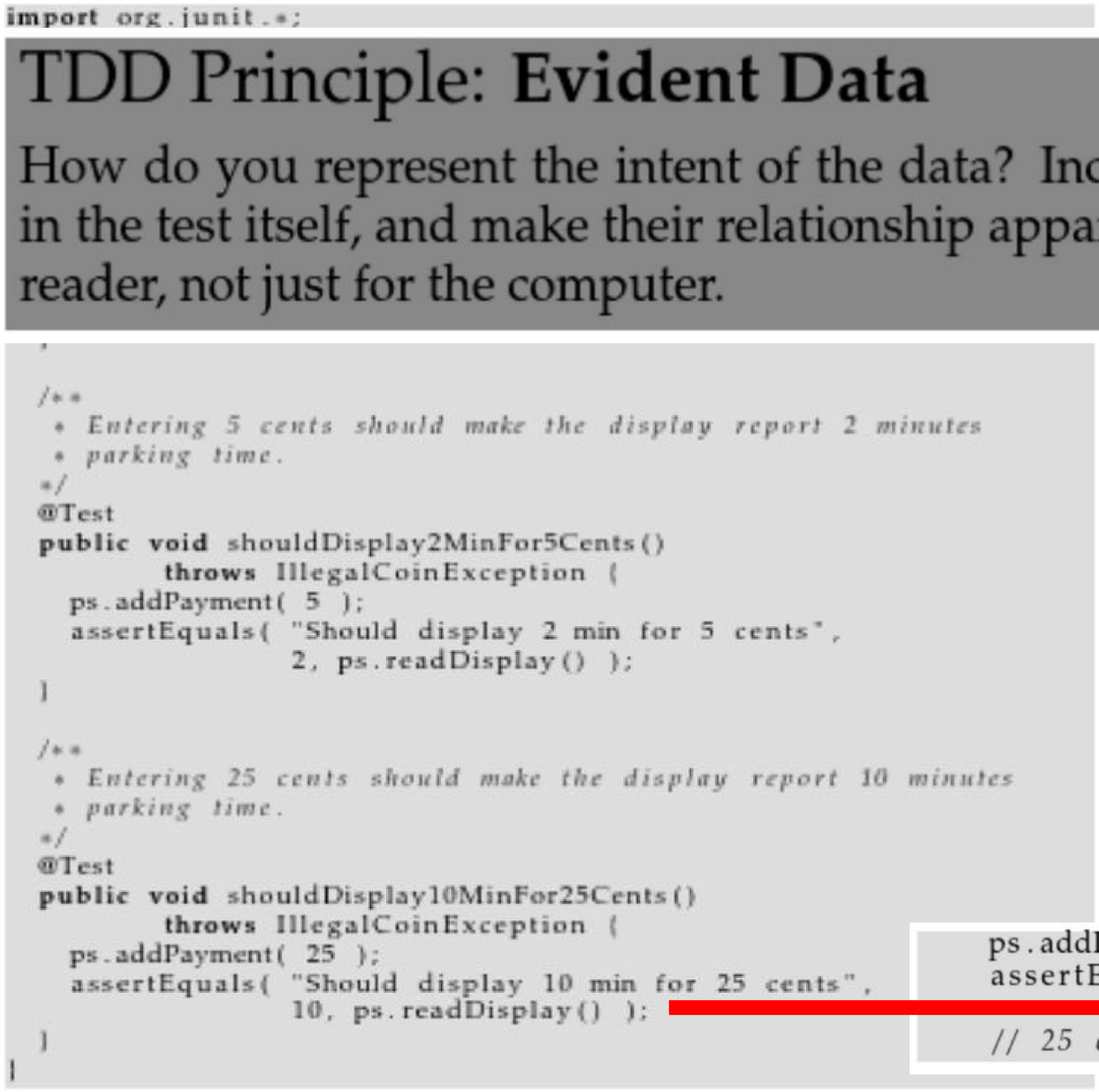


- ✻ ~~accept legal coin~~
- ✻ reject illegal coin, exception
- ✻ ~~5 cents should give 2 minutes parking time.~~
- ✻ ~~readDisplay~~
- ✻ buy produces valid receipt
- ✻ cancel resets pay station
- ✻ *25 cents = 10 minutes*
- ✻ enter two or more legal coins



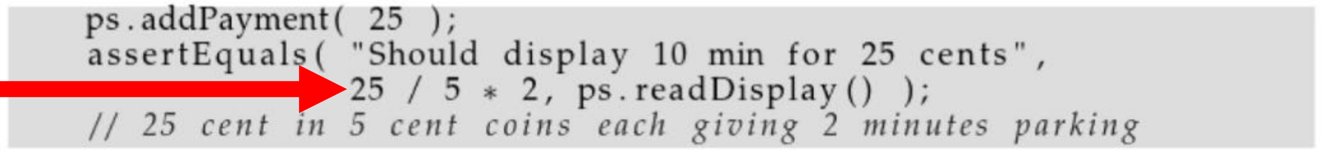
TDD Principle: **Evident Data**

How do you represent the intent of the data? Include expected and actual results in the test itself, and make their relationship apparent. You are writing tests for the reader, not just for the computer.



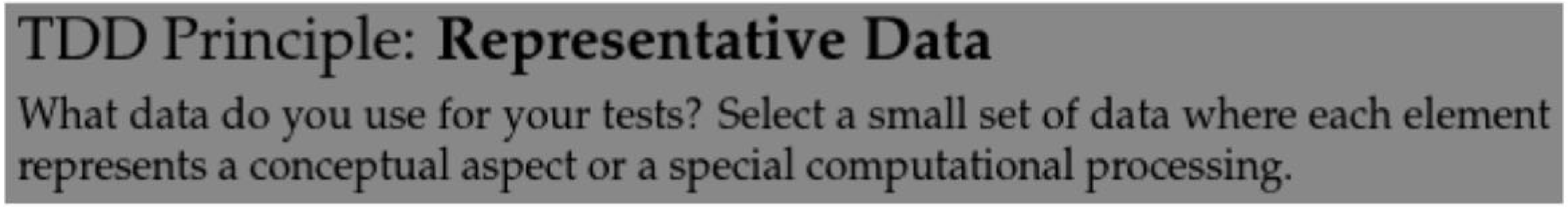
```
import org.junit.*;
...
  /**
   * Entering 5 cents should make the display report 2 minutes
   * parking time.
   */
  @Test
  public void shouldDisplay2MinFor5Cents()
          throws IllegalCoinException {
    ps.addPayment( 5 );
    assertEquals( "Should display 2 min for 5 cents",
                  2, ps.readDisplay() );
  }

  /**
   * Entering 25 cents should make the display report 10 minutes
   * parking time.
   */
  @Test
  public void shouldDisplay10MinFor25Cents()
          throws IllegalCoinException {
    ps.addPayment( 25 );
    assertEquals( "Should display 10 min for 25 cents",
                  10, ps.readDisplay() );
  }
}
```

Change code or add comments (why not both?)



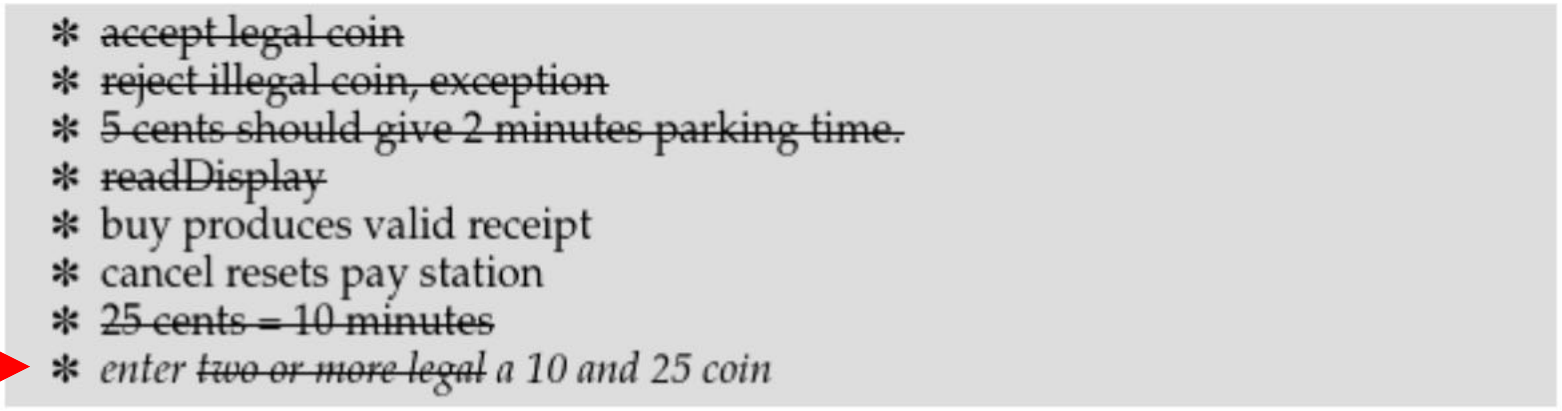
```
ps.addPayment( 25 );
assertEquals( "Should display 10 min for 25 cents",
               25 / 5 * 2, ps.readDisplay() );
// 25 cent in 5 cent coins each giving 2 minutes parking
```

Case: PayStation



TDD Principle: **Representative Data**

What data do you use for your tests? Select a small set of data where each element represents a conceptual aspect or a special computational processing.



- ~~accept legal coin~~
- ~~reject illegal coin, exception~~
- ~~5 cents should give 2 minutes parking time.~~
- ~~readDisplay~~
- buy produces valid receipt
- cancel resets pay station
- ~~25 cents = 10 minutes~~
- *enter ~~two or more legal~~ a 10 and 25 coin*

Case: PayStation

- ✱ ~~accept legal coin~~
- ✱ ~~reject illegal coin, exception~~
- ✱ ~~5 cents should give 2 minutes parking time.~~
- ✱ ~~readDisplay~~
- ✱ ~~buy for 40 cents produces valid receipt~~
- ✱ cancel resets pay station
- ✱ ~~25 cents = 10 minutes~~
- ✱ ~~enter a 10 and 25 coin~~
- ✱ *receipt can store values*
- ✱ *buy for 100 cents*

1. Add a test

TDD Principle: Assert First

When should you write the asserts? Try writing them first.

```
@Test
public void shouldStoreTimeInReceipt() {
  ...
  assertEquals( "Receipt can store 30 minute value",
                30, receipt.value() );
}
```

```
@Test
public void shouldStoreTimeInReceipt() {
  Receipt receipt = new ReceiptImpl(30);
  assertEquals( "Receipt can store 30 minute value",
                30, receipt.value() );
}
```

Case: PayStation

```
@Test
public void shouldStoreTimeInReceipt() {
  Receipt receipt = new ReceiptImpl(30);
  assertEquals( "Receipt can store 30 minute value",
                30, receipt.value() );
}
```

- ✻ ~~accept legal coin~~
- ✻ ~~reject illegal coin, exception~~
- ✻ ~~5 cents should give 2 minutes parking time.~~
- ✻ ~~readDisplay~~
- ✻ ~~buy for 40 cents produces valid receipt~~
- ✻ cancel resets pay station
- ✻ ~~25 cents = 10 minutes~~
- ✻ ~~enter a 10 and 25 coin~~
- ✻ *receipt can store values*
- ✻ *buy for 100 cents*

3. Make a little change

ReceiptImpl.java

```
/** Implementation of Receipt.
*/

public class ReceiptImpl implements Receipt {
  private int value;
  public ReceiptImpl(int value) { this.value = value; }
  public int value() { return value;}
}
```

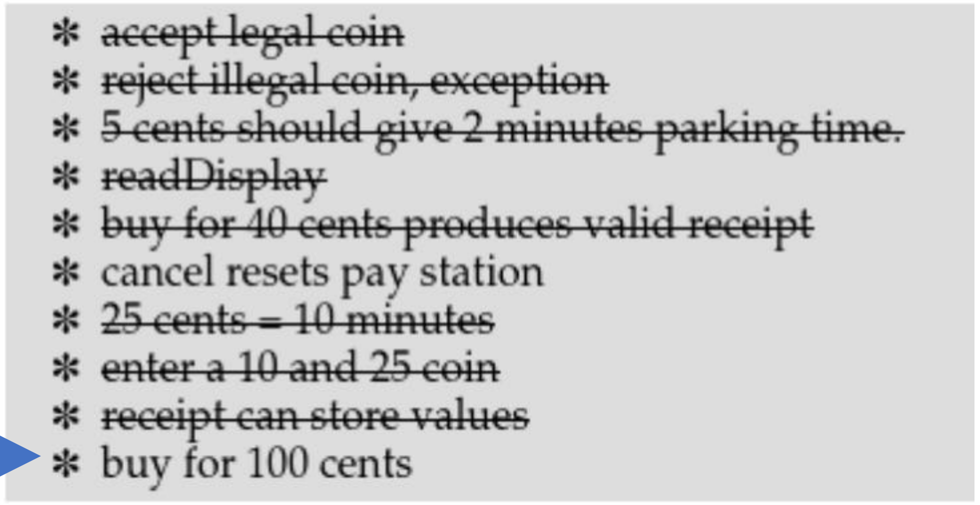
TDD Principle: Obvious Implementation

How do you implement simple operations? Just implement them.

When implementation is simple, we can skip fake-it and triangulation

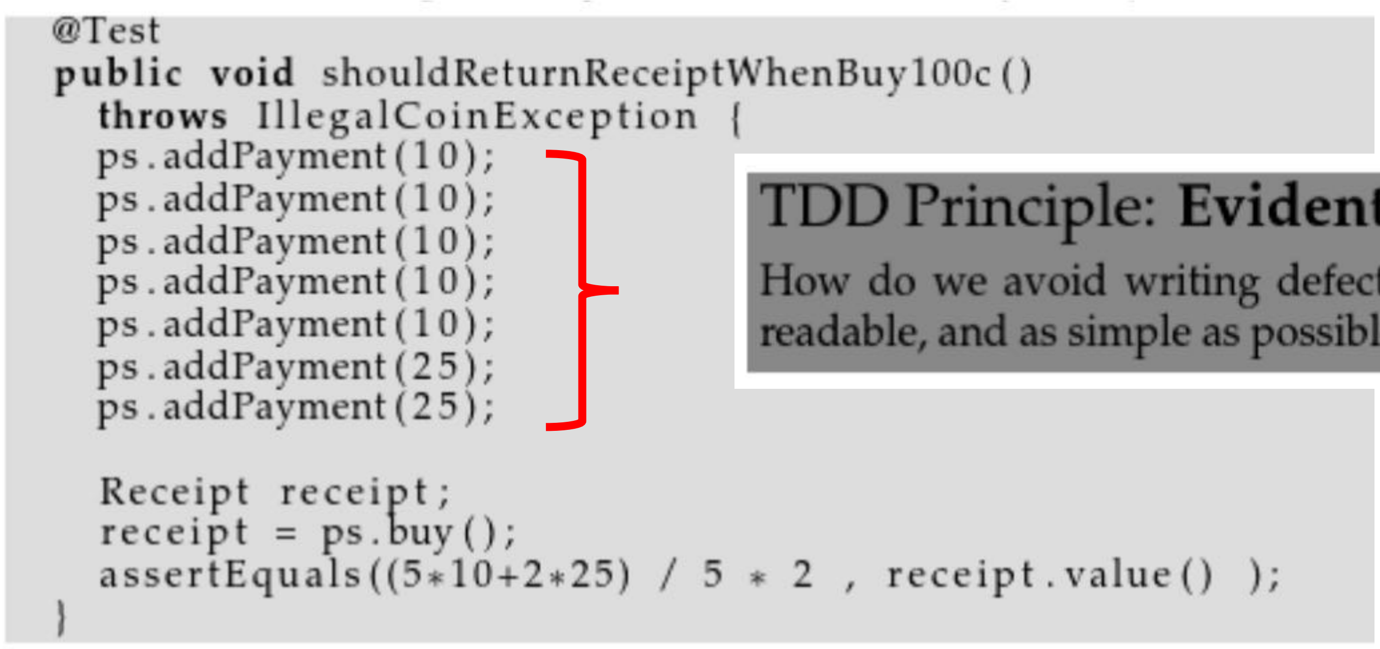
4. See the tests pass

Case: PayStation

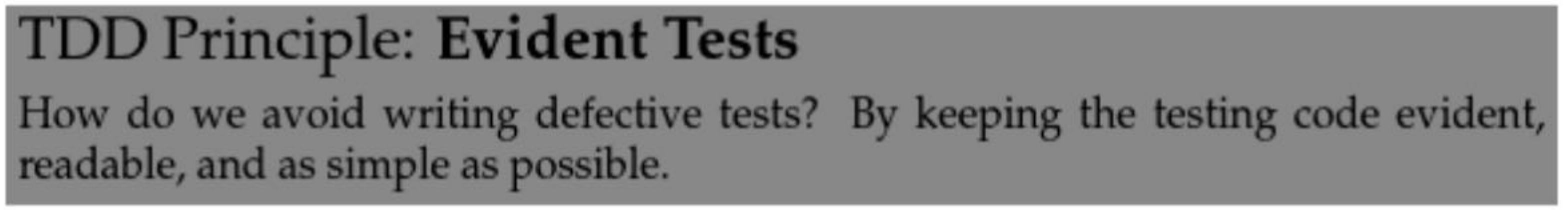


* ~~accept legal coin~~
* ~~reject illegal coin, exception~~
* ~~5 cents should give 2 minutes parking time.~~
* ~~readDisplay~~
* ~~buy for 40 cents produces valid receipt~~
* cancel resets pay station
* ~~25 cents = 10 minutes~~
* ~~enter a 10 and 25 coin~~
* ~~receipt can store values~~
* buy for 100 cents

1. Add a test



```
@Test
public void shouldReturnReceiptWhenBuy100c()
  throws IllegalCoinException {
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(10);
  ps.addPayment(25);
  ps.addPayment(25);

  Receipt receipt;
  receipt = ps.buy();
  assertEquals((5*10+2*25) / 5 * 2 , receipt.value() );
}
```



TDD Principle: Evident Tests

How do we avoid writing defective tests? By keeping the testing code evident, readable, and as simple as possible.

Test cases should ideally only contain assignments, method calls, and assertions (maybe very simple loops, definitely not recursion). Private helper methods are okay

HotCiv: Test-Driven Development

The more your testcases only use the given Game, City, Unit interfaces, the more stable your test cases will be against refactoring inner data structures!

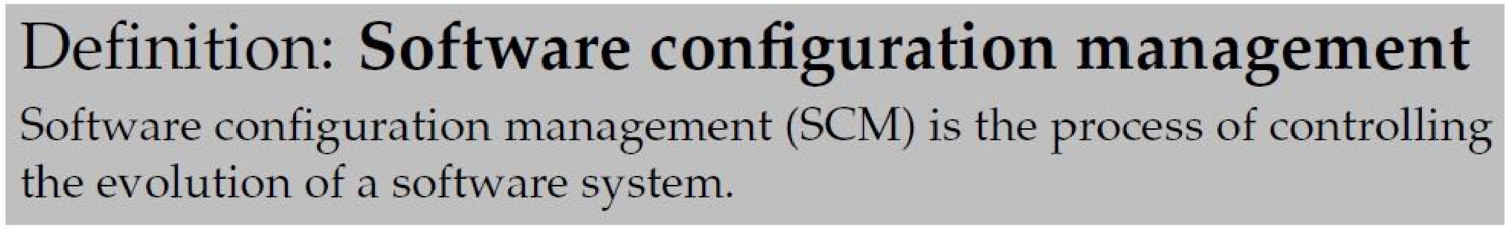
So, do this:

- game.getCityAt(p)

Not this:

- ((GameImpl) game).getInternalCityHashmap.get(p)

Configuration Management



Definition: **Software configuration management**

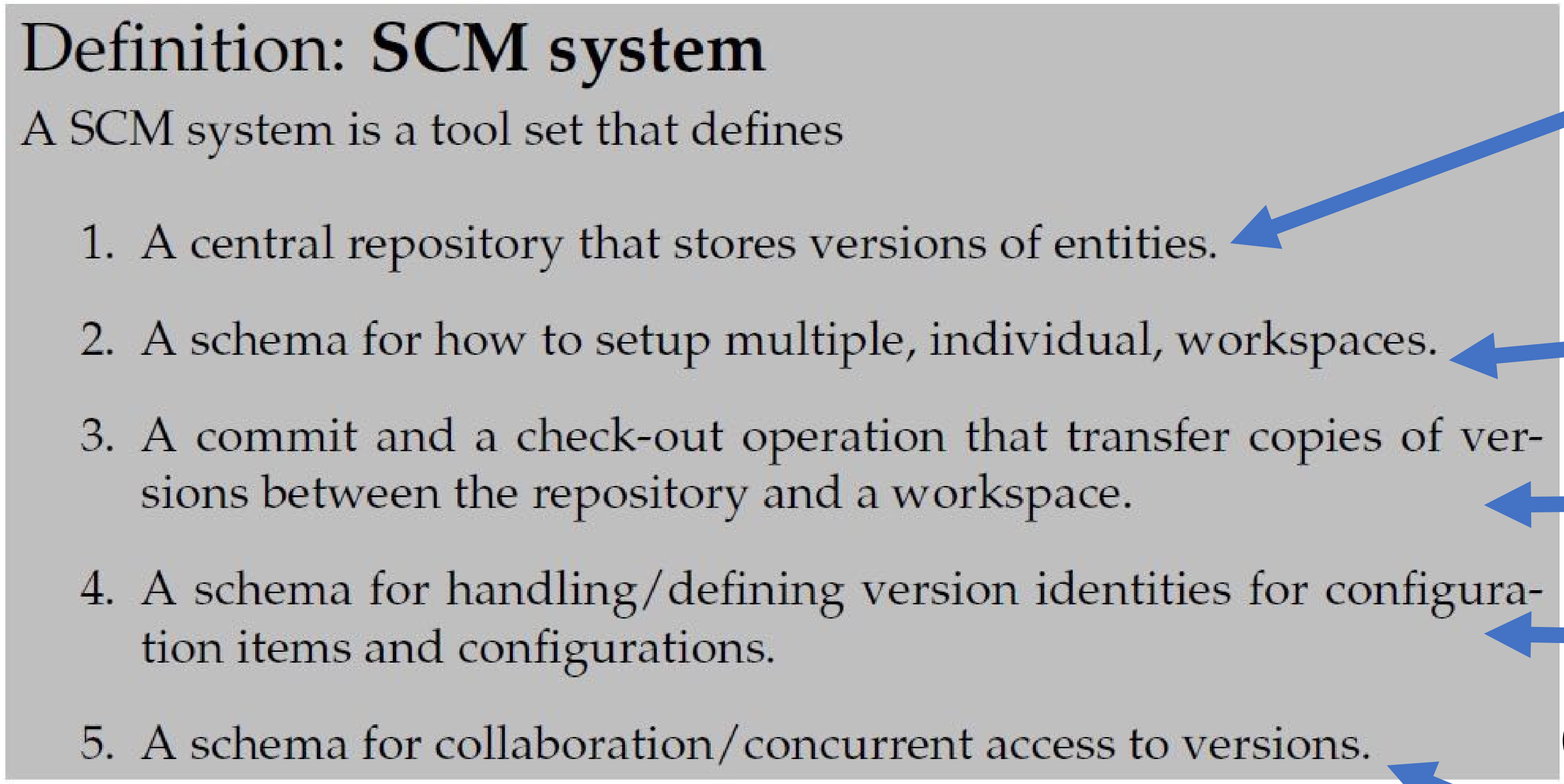
Software configuration management (SCM) is the process of controlling the evolution of a software system.

Configuration Management helps to solve many problems in development!

- Team collaboration on large code bases
- Developing multiple features in parallel
- Managing release vs development versions
- Managing breaking changes
- Iterative software development

We will use Git

Version Control: Git



Definition: **SCM system**

A SCM system is a tool set that defines

1. A central repository that stores versions of entities.
2. A schema for how to setup multiple, individual, workspaces.
3. A commit and a check-out operation that transfer copies of versions between the repository and a workspace.
4. A schema for handling/defining version identities for configuration items and configurations.
5. A schema for collaboration/concurrent access to versions.

.git

git clone

git commit/checkout

Commit IDs (hashes)

Optimistic concurrency
clone/checkout
merge

Git is a **distributed** software configuration management system

- Every workspace holds a complete copy of the repository (repo)

Version Control: Git

To be included in a commit, a new or modified file in workspace must be added to the Git **staging area** with **git add**

Workflow:

- Working locally, modify README.md
- See a list of changes: **git status**
- Add files to the staging area: **git add README.m**d
- Commit to the local repo: **git commit –m ”modified readme”**
- Merge local commits into remote repo: **git push**

Git Commands

git status

git add <. or file names>

git commit

git push

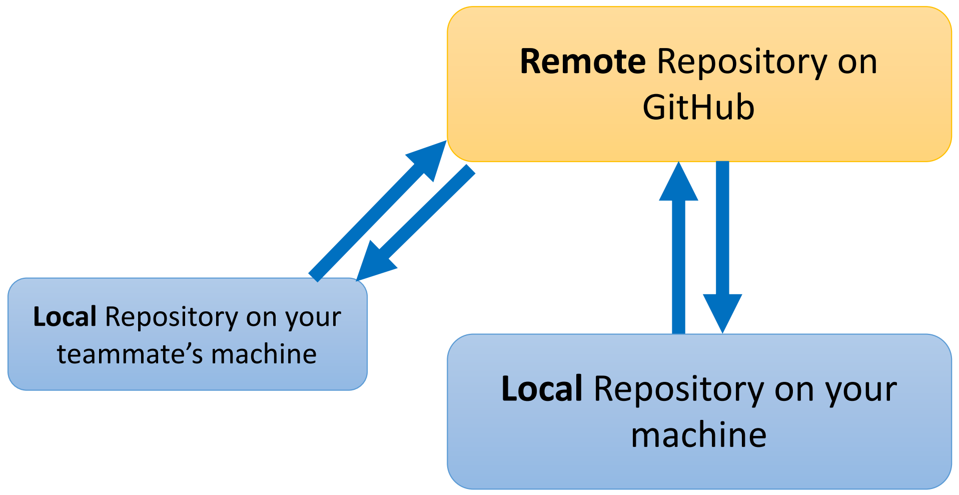
git pull

git tag

git log

git checkout

git diff


Remote Repository on GitHub
Local Repository on your teammate's machine
Local Repository on your machine

Version Control: Best Practices

Commit **related** changes

- Fixing two bugs should lead to two commits

Commit often

- ‘Take small steps’, break big into small, one step at a time
- Keeps a safe version to checkout in case of ‘Do Over’

Use **informative commit messages**! You may need to search through the log for a previous commit to return to

Push often to sync your work

- Can re-clone your remote repo if something goes wrong

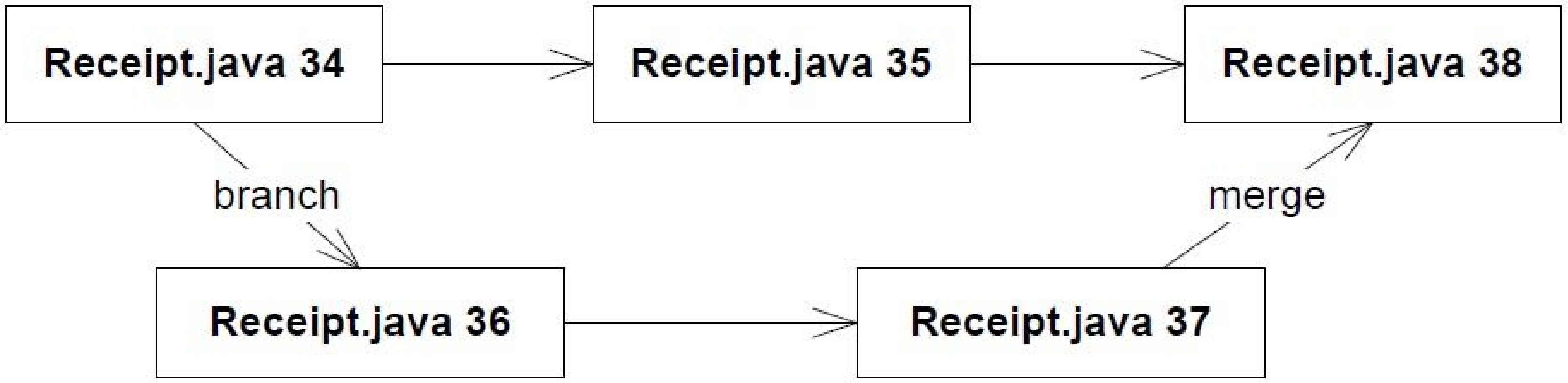
Avoid pushing broken commits!

- Can commit broken builds locally if changing tasks, to preserve work in progress
- Pushed commits should reflect a finished step/feature/bugfix
    - **All tests should pass in pushed code**

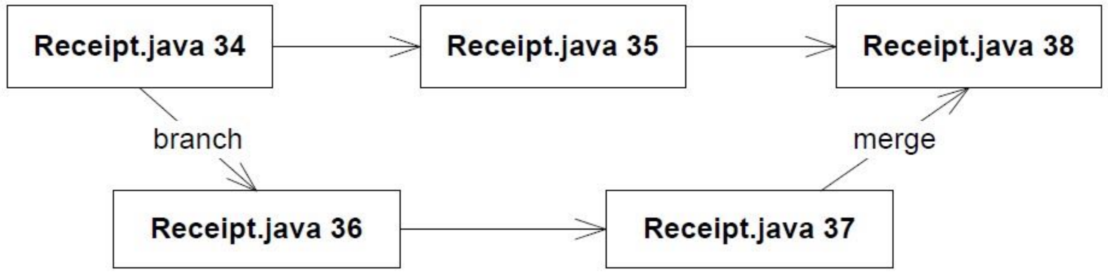
Branching

Definition: Branch

A branch is a point in the version graph where a version is ancestor to two or more descendant versions.



Branching



git checkout -b <new branch name>

- Create and checkout a new branch off of the current one
- Any changes and commits will be on the new branch

git merge <branch name>

- Merge all commits from the specified branch into the current branch

git checkout <branch name>

- Checkout the specified branch (when branch already exists)

git branch -a

- Show all branches *including* all on the origin that you do not have currently

git branch -d <branch name>

- Delete a branch (do after merging feature branches to clean up the branch list)
- **History is preserved**
- git push origin :<branch name> to also delete the branch on the remote (make sure no one else needs it!)

Branching Models: Single Release Branch

Single Release Branch

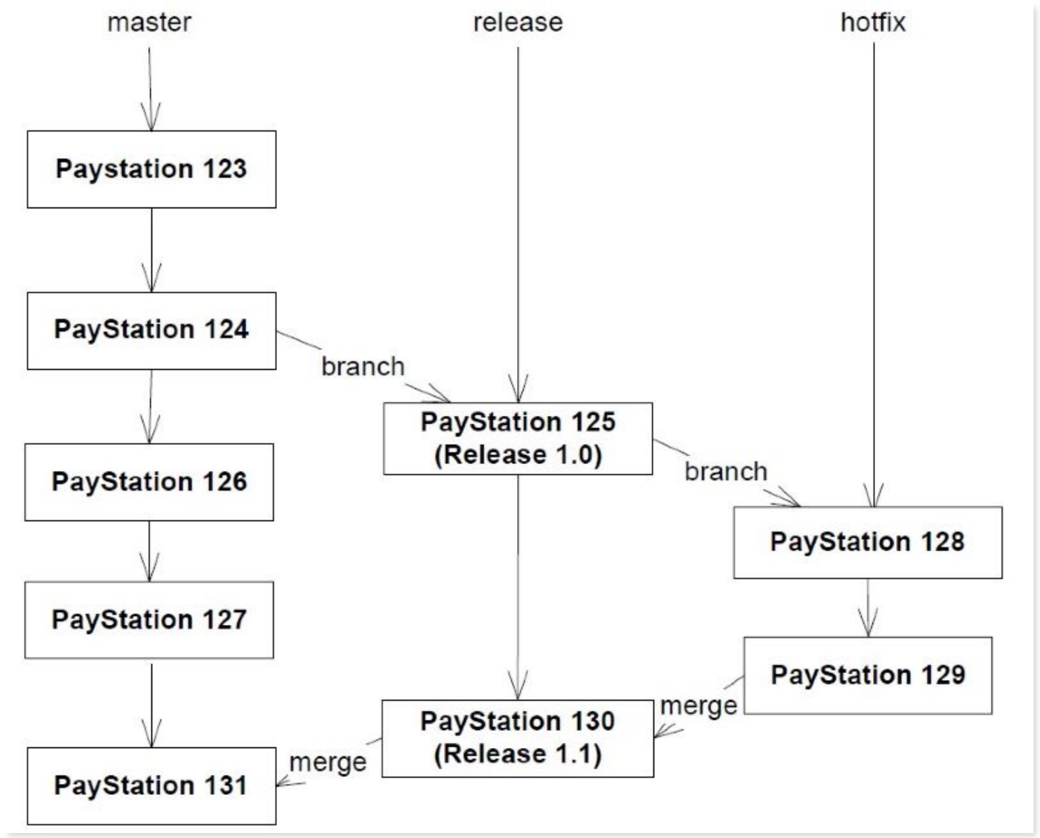
- Dedicated branch for releases
- Hotfixing must be done on separate branch and merged back

Pros

- Get the latest release is just 'git checkout release' which is easy
- Fewer branches (only 'release' and 'hotfix' used for releasing)

Cons

- You need a separate branch for hotfixing
  - Branch/Merge over into hotfix, make changes, merge back
- Hotfixing release 1 after release 2 **will** require a new branch
  - Or you will mix the two!



Branching Models: Major Release Branches

Major Release Branches

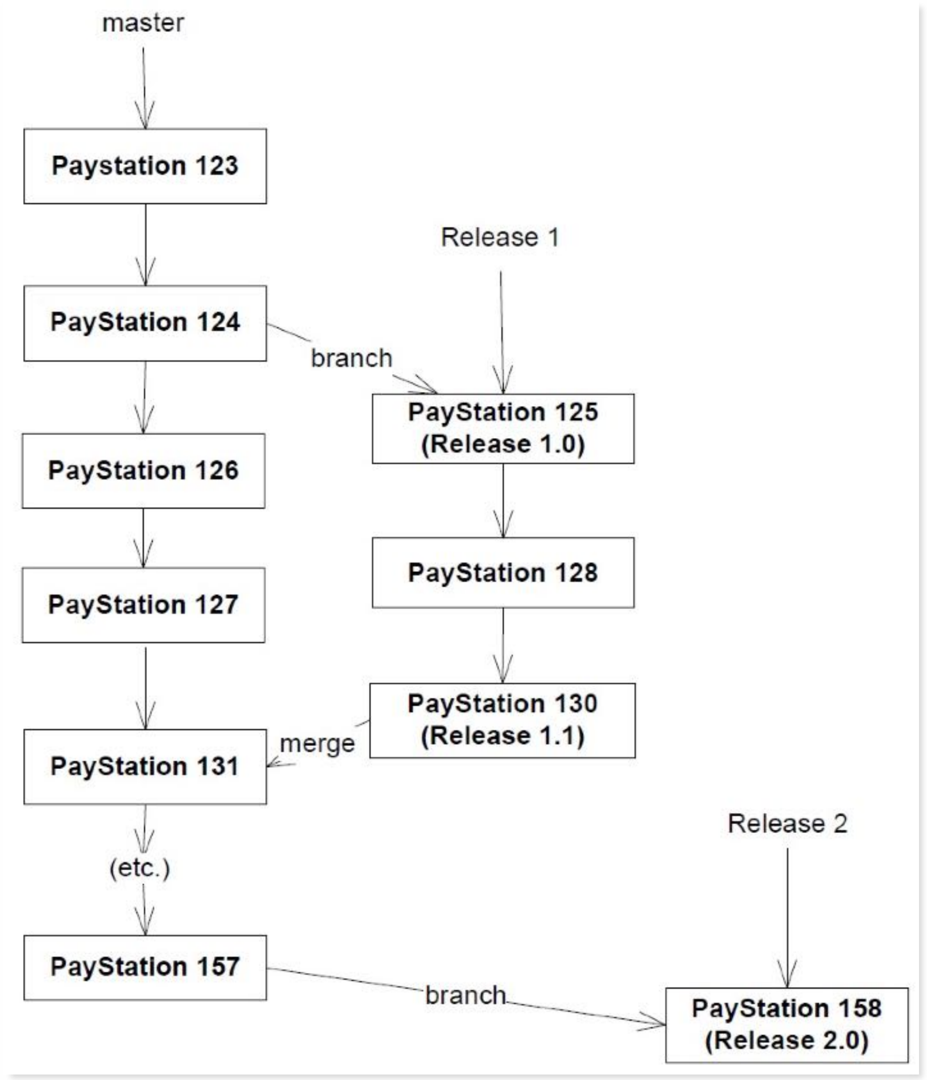
- Each major release gives rise to new branch

Pro

- Each major release has a branch, allows hotfixing each release with ease

Con

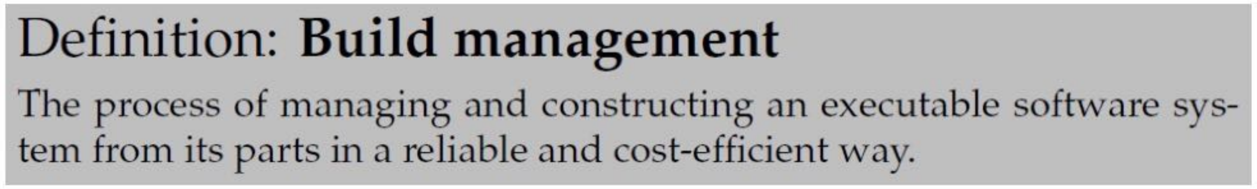
- Many releases means many branches to maintain



Build Management

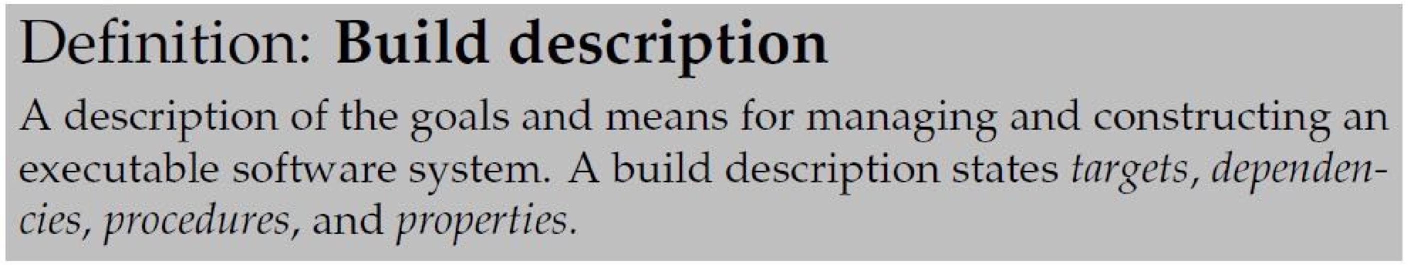
As a project gets more complicated, how do we ensure we build the same way each time?

→ Automated build management



Definition: **Build management**

The process of managing and constructing an executable software system from its parts in a reliable and cost-efficient way.



Definition: **Build description**

A description of the goals and means for managing and constructing an executable software system. A build description states *targets*, *dependencies*, *procedures*, and *properties*.

Description/Script

Build Management: Script

Targets: tasks/goals, like “compile all source code files”

Dependencies: tasks can depend on other tasks (e.g., you must compile all source code before executing, so the execution target **depends on** the compilation target); or external libraries required for the build

Procedures: associated with the targets; describe how to meet the goal of the target

- For example, the compile goal must have an associated procedure that describes the steps necessary to compile all source files (e.g., call javac on all files)

Properties: variables and constants that you can assign and use in your procedures; used to improve readability in the build script

Build Management: Gradle

How does it work?

Convention

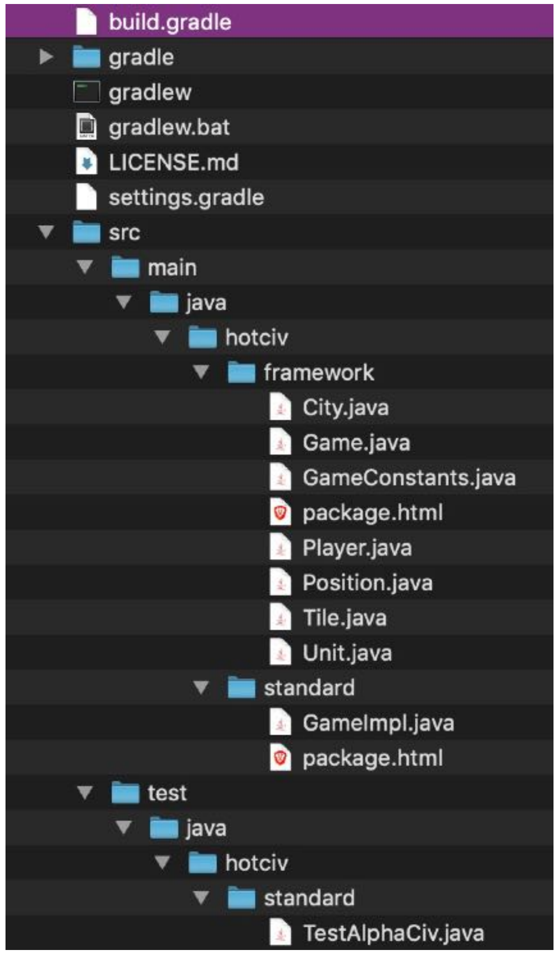
- You must put your code in the right folders!
  - **src/main/java/HERE** ← Source/production code
  - **src/test/java/HERE** ← Test code
- Predefined tasks
  - Like ‘build’, ‘test’, …

Plugins

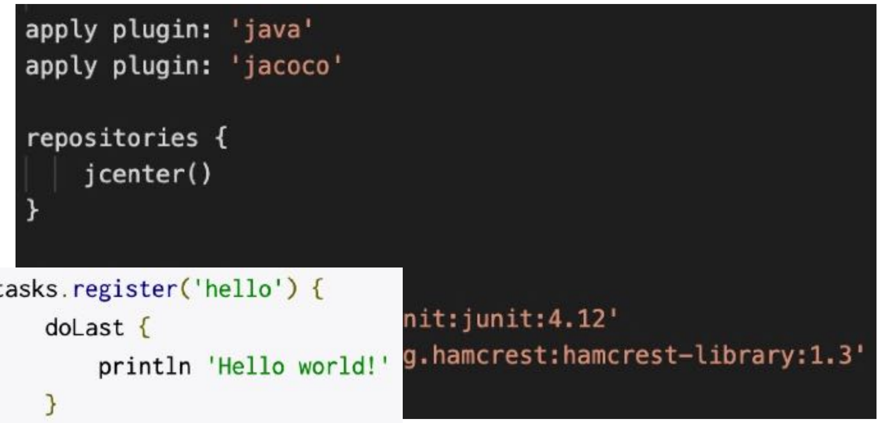
- Define tasks, conventions
  https://docs.gradle.org/current/userguide/plugin_reference.html

Custom tasks in Groovy or Kotlin, if needed
https://docs.gradle.org/current/userguide/tutorial_using_tasks.html

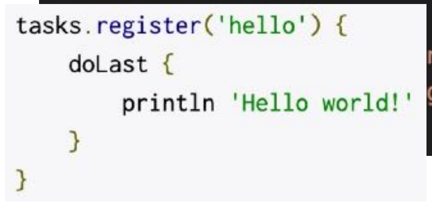
(not needed for this class)



```
build.gradle
▶ gradle
gradlew
gradlew.bat
LICENSE.md
settings.gradle
▼ src
  ▼ main
    ▼ java
      ▼ hotciv
        ▼ framework
            City.java
            Game.java
            GameConstants.java
            package.html
            Player.java
            Position.java
            Tile.java
            Unit.java
        ▼ standard
            GameImpl.java
            package.html
  ▼ test
    ▼ java
      ▼ hotciv
        ▼ standard
            TestAlphaCiv.java
```



```
apply plugin: 'java'
apply plugin: 'jacoco'

repositories {
    jcenter()
}
nit:junit:4.12'
g.hamcrest:hamcrest-library:1.3'
```

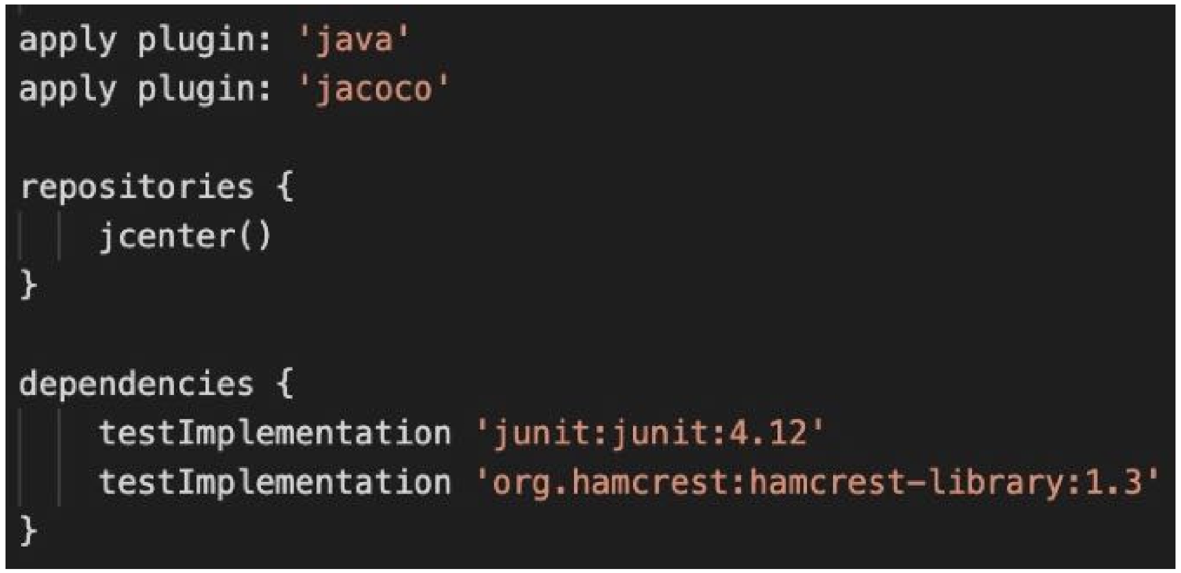


```
tasks.register('hello') {
    doLast {
        println 'Hello world!'
    }
}
```

Dependency Management: Gradle

Gradle is also a **dependency-management tool.**

Ex: hamcrest

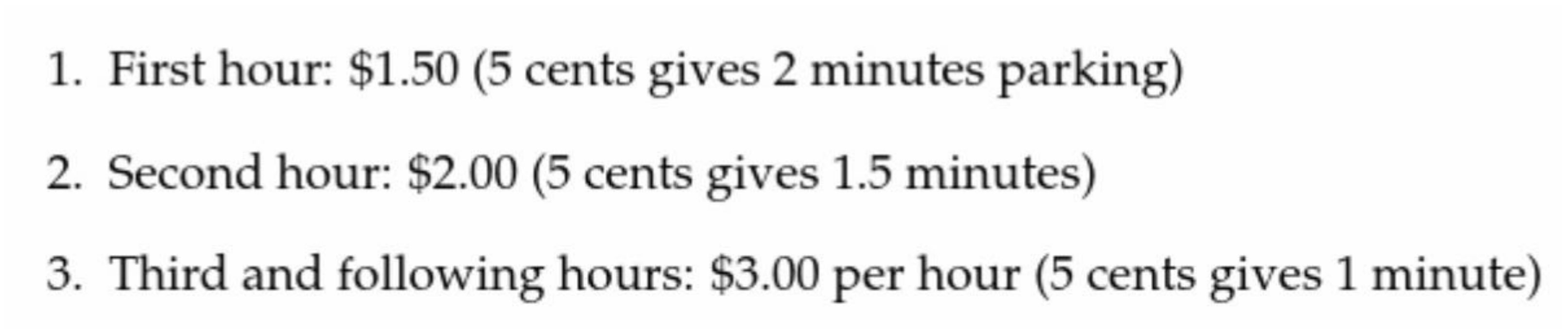


```
apply plugin: 'java'
apply plugin: 'jacoco'

repositories {
    jcenter()
}

dependencies {
    testImplementation 'junit:junit:4.12'
    testImplementation 'org.hamcrest:hamcrest-library:1.3'
}
```

Gradle will download 'org.hamcrest….:1.3' from JCenter and set the classpath accordingly

BetaTown: New Customer Requirements!

Progressive price model



1. First hour: $1.50 (5 cents gives 2 minutes parking)
2. Second hour: $2.00 (5 cents gives 1.5 minutes)
3. Third and following hours: $3.00 per hour (5 cents gives 1 minute)

Maybe we will have more requests for different pricing models ???

How can we handle these two products (and future variants)?

Variability

Problem (formulation 1): We need to develop and maintain two variants of the software system in a way that introduces the least cost and defects.

Problem (formulation 2): How do I introduce having two different behaviors of the rate calculation variability point such that both the cost and the risk of introducing defects are low?

```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5: break;
  case 10: break;
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar += coinValue;
  timeBought = insertedSoFar / 5 * 2;
}
```

Definition: Variability point

A variability point is a well defined section of the production code whose behavior it should be possible to vary.

Variability

Model 1: **Copy**

- Make a copy of the source tree - one copy for AlphaTown, one for BetaTown

Model 2: **Parameterization**

- Throw in some ‘if’ statements to check whether AlphaTown or BetaTown

Model 3: **Polymorphism**

- Use inheritance, subclass PayStationImpl, override the rate calculation

Model 4: **Composition**

- Factor out rate calculation responsibility into an interface, create multiple concrete rate calculation classes, pay station delegates to the appropriate rate calculation object

Model 1: Copy Source Tree

Pros:

- Simple!
  - No special skill set required in developer team
  - Easy to explain to new developers
- Fast!
  - Copy, paste, modify
- Decoupled!
  - Defects introduced in variant 2 does **not** reduce reliability of variant 1
  - Easy to distinguish variants

Cons:

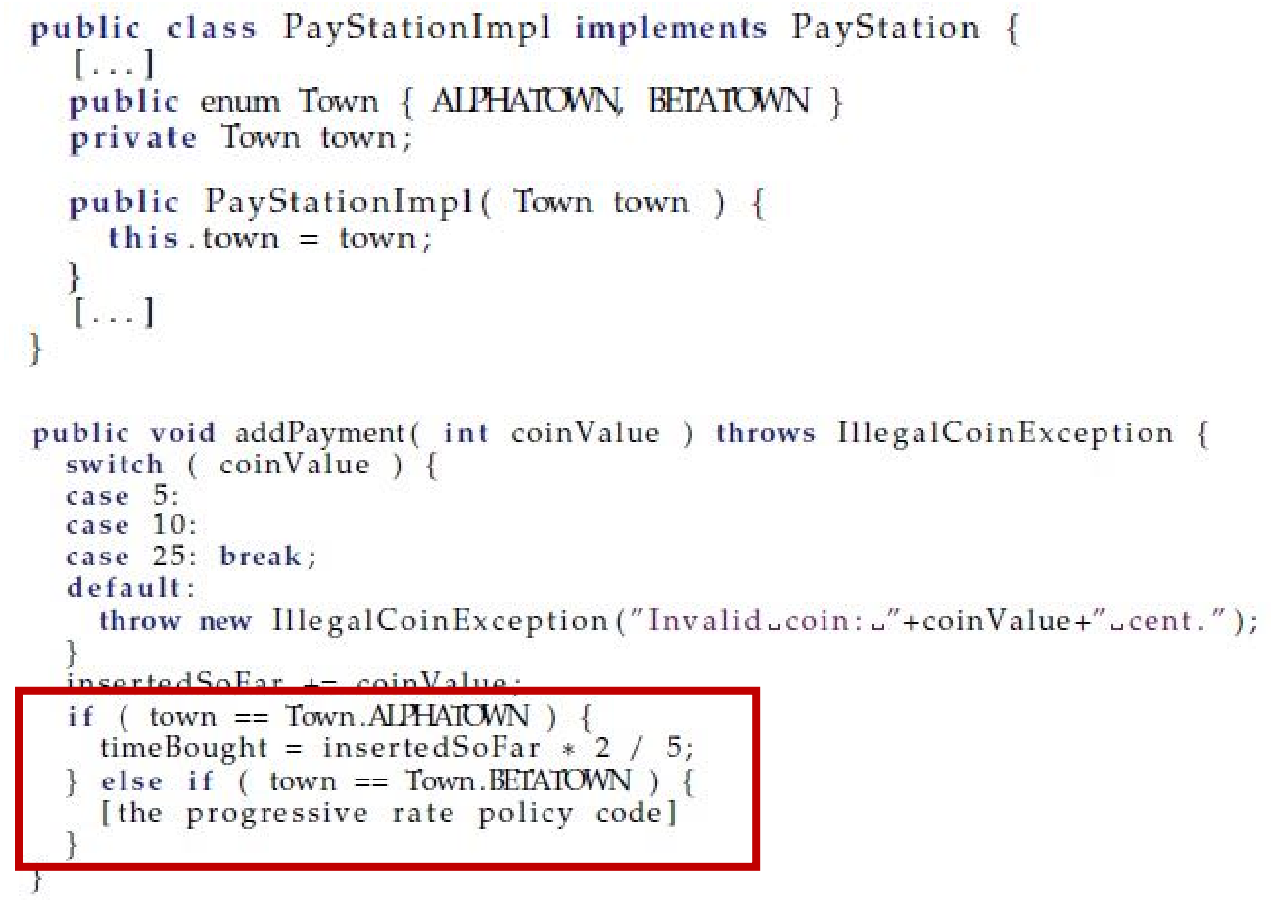
- **Multiple maintenance problem**
  - Changes in common code must be propagated to all copies
- Example:
  - 4 pay station variants (different rate policies) = 4 source trees
  - Feature request: pay station keeps track of earnings ☹
  - Fix the same bug in nearly identical production code bases at the same time ☹
- Typically, variants drift apart, becoming different products instead of variants...

Model 2: Parametric Solution

Variability point is in a single “behavioural unit” in the addPayment method.

Make a conditional statement there

- Introduce a “town” parameter
- Switch on the parameter each time town-specific behaviour is needed



```
public class PayStationImpl implements PayStation {
  [...]
  public enum Town { ALPHATOWN, BETATOWN }
  private Town town;

  public PayStationImpl( Town town ) {
    this.town = town;
  }
  [...]
}

public void addPayment( int coinValue ) throws IllegalCoinException {
  switch ( coinValue ) {
  case 5:
  case 10:
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue+" cent.");
  }
  insertedSoFar += coinValue;
  if ( town == Town.ALPHATOWN ) {
    timeBought = insertedSoFar * 2 / 5;
  } else if ( town == Town.BETATOWN ) {
    [the progressive rate policy code]
  }
}
```

Model 2: Parametric Solution

Pros:

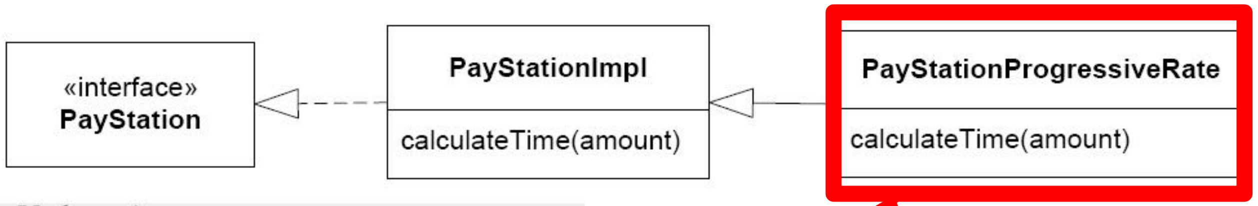
- Simple!
  - A conditional statement is easy to understand for any skill level developer team
- Avoids multiple maintenance problem!
  - One code base
  - Easier to fix defects in common behavior
  - Easier to add common features

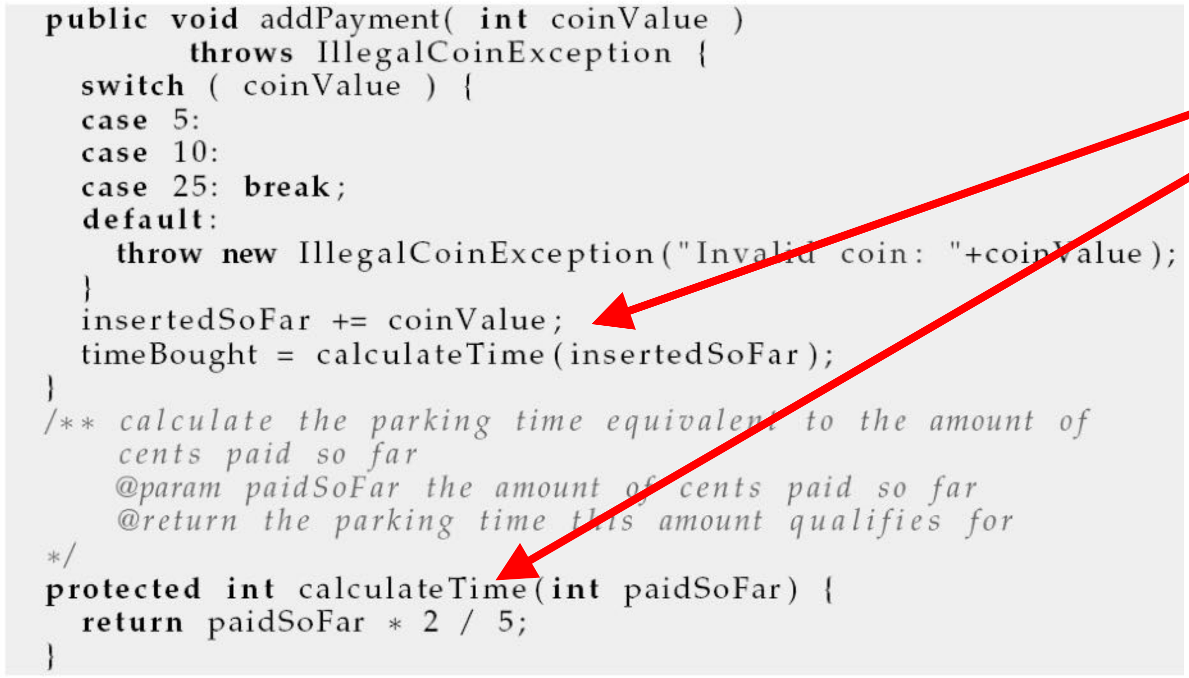
Cons:

- Reliability concerns
  - Risks of **change by modification**
- Analyzability concerns
  - Code bloat, switch creep
- Responsibility erosion
  - “feature creep”
- Composition problem
  - A rate model that is a combination of existing ones leads to code duplication

Model 3: Polymorphic Solution

Subclass and override!



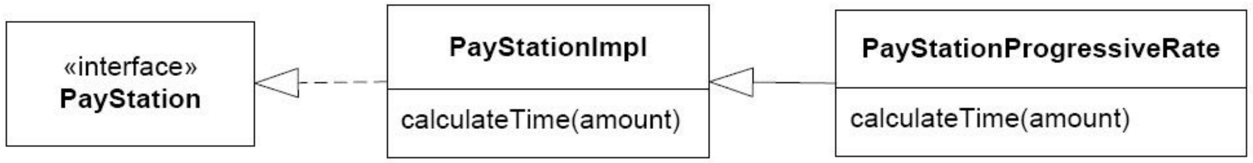


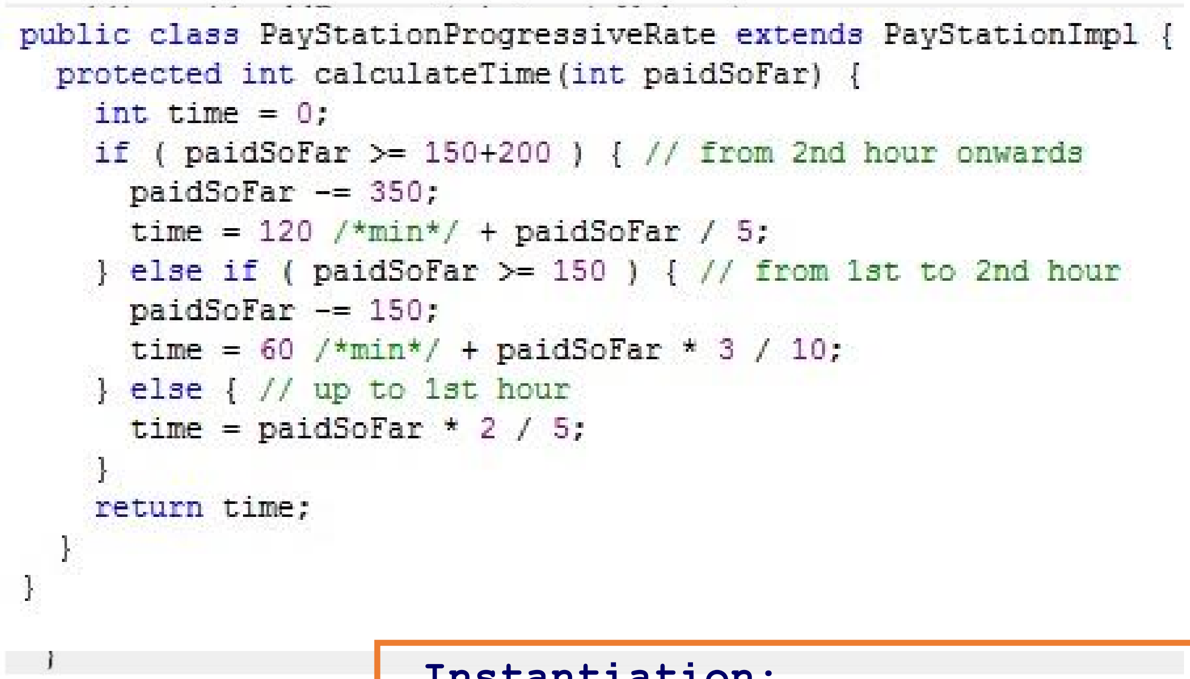
```
public void addPayment( int coinValue )
        throws IllegalCoinException {
  switch ( coinValue ) {
  case 5:
  case 10:
  case 25: break;
  default:
    throw new IllegalCoinException("Invalid coin: "+coinValue);
  }
  insertedSoFar += coinValue;
  timeBought = calculateTime(insertedSoFar);
}
/** calculate the parking time equivalent to the amount of
    cents paid so far
    @param paidSoFar the amount of cents paid so far
    @return the parking time this amount qualifies for
*/
protected int calculateTime(int paidSoFar) {
  return paidSoFar * 2 / 5;
}
```

Refactor to introduce calculateTime method

Model 3: Polymorphic Solution

Subclass and override!





```
public class PayStationProgressiveRate extends PayStationImpl {
  protected int calculateTime(int paidSoFar) {
    int time = 0;
    if ( paidSoFar >= 150+200 ) { // from 2nd hour onwards
      paidSoFar -= 350;
      time = 120 /*min*/ + paidSoFar / 5;
    } else if ( paidSoFar >= 150 ) { // from 1st to 2nd hour
      paidSoFar -= 150;
      time = 60 /*min*/ + paidSoFar * 3 / 10;
    } else { // up to 1st hour
      time = paidSoFar * 2 / 5;
    }
    return time;
  }
}
```

```
Instantiation:
PayStation ps =
   new PayStationProgressiveRate();
```

Model 3: Polymorphic Solution

Pros:

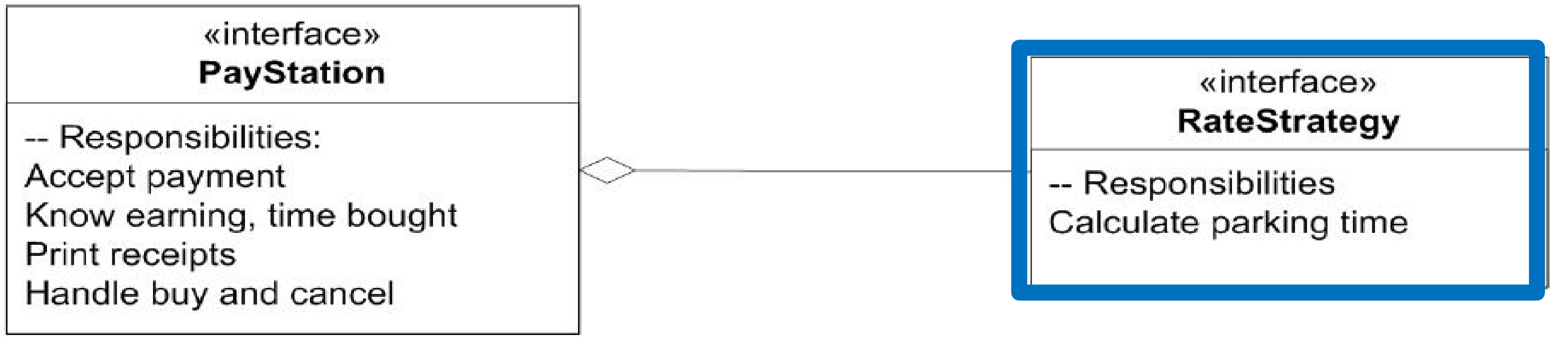
- Avoid multiple maintenance
- Avoid reliability concerns
- Code analyzability

Cons:

- Increased number of classes
- Inheritance spent on one type of variation
- Compile-time binding
- Reuse across variants is difficult

Model 4: Composition

Method: Divide responsibilities!
→ Objects collaborate to provide the required behavior in combination



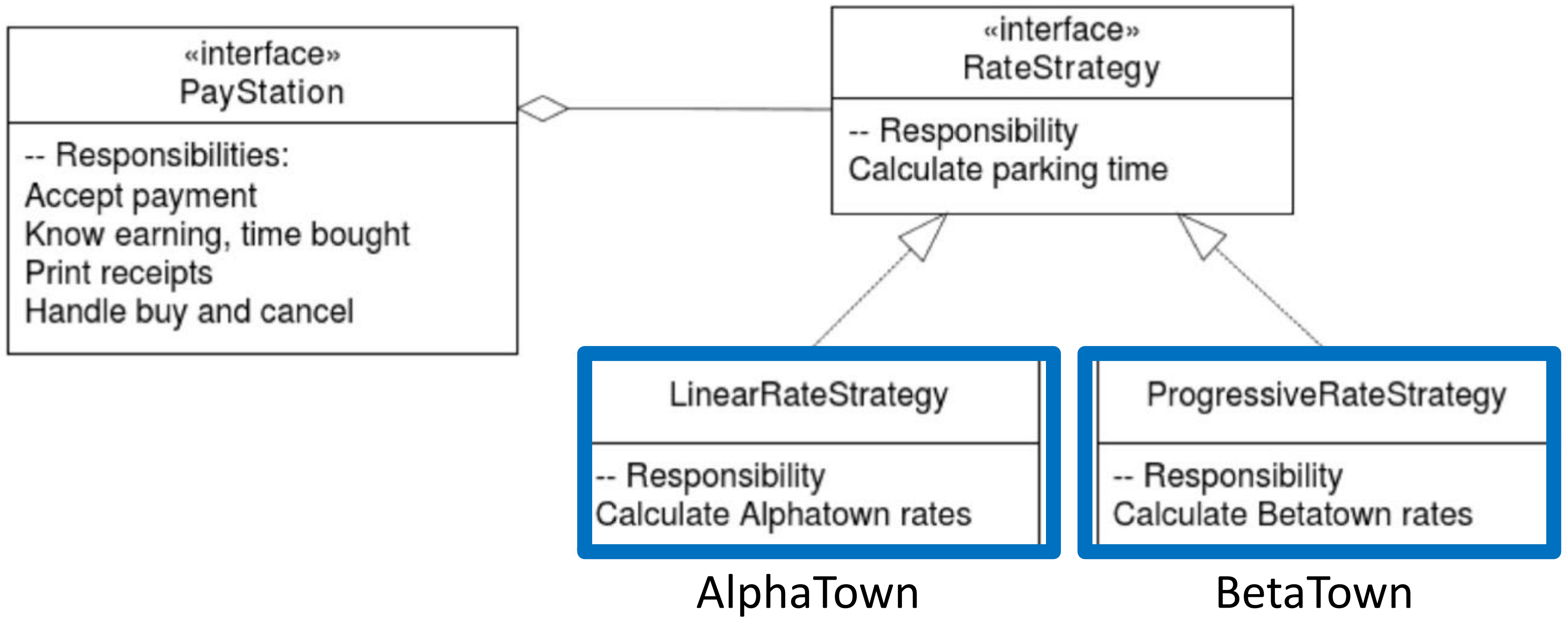
```
[...]
insertedSoFar += coinValue;
timeBought = someOtherObject.calculateTime(insertedSoFar);
```

Put the responsibility in its own abstraction / object

Model 4: Composition

Definition: Delegation

In delegation, two objects collaborate to satisfy a request or fulfill a responsibility. The behavior of the receiving object is partially handled by a subordinate object, called the **delegate**.



Model 4: Composition

Pros:

- Separation of responsibilities
  - Easier testing, debugging
- Variant selection is localized
- Combinatorial
  - Contrast with inheritance
  - Doesn’t affect variations in other behavior
- **Reliability**
  - New classes for new rate policies
- **Analyzability**
  - No conditional statements
- **Run-time binding**
  - Can change rate policy while running

Cons:

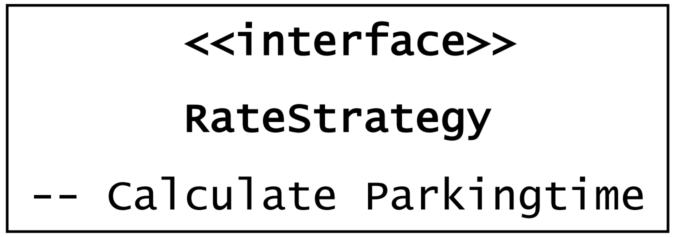
- Increased number of interfaces, objects
- Client objects must be aware of strategies

3-1-2 Process (More when we get to Compositional Design)

③ **Find Variability:** Identify some behavior that is likely to change…

- Rate policy

① **Use an Interface:** State a responsibility that covers this behavior and express it in an interface



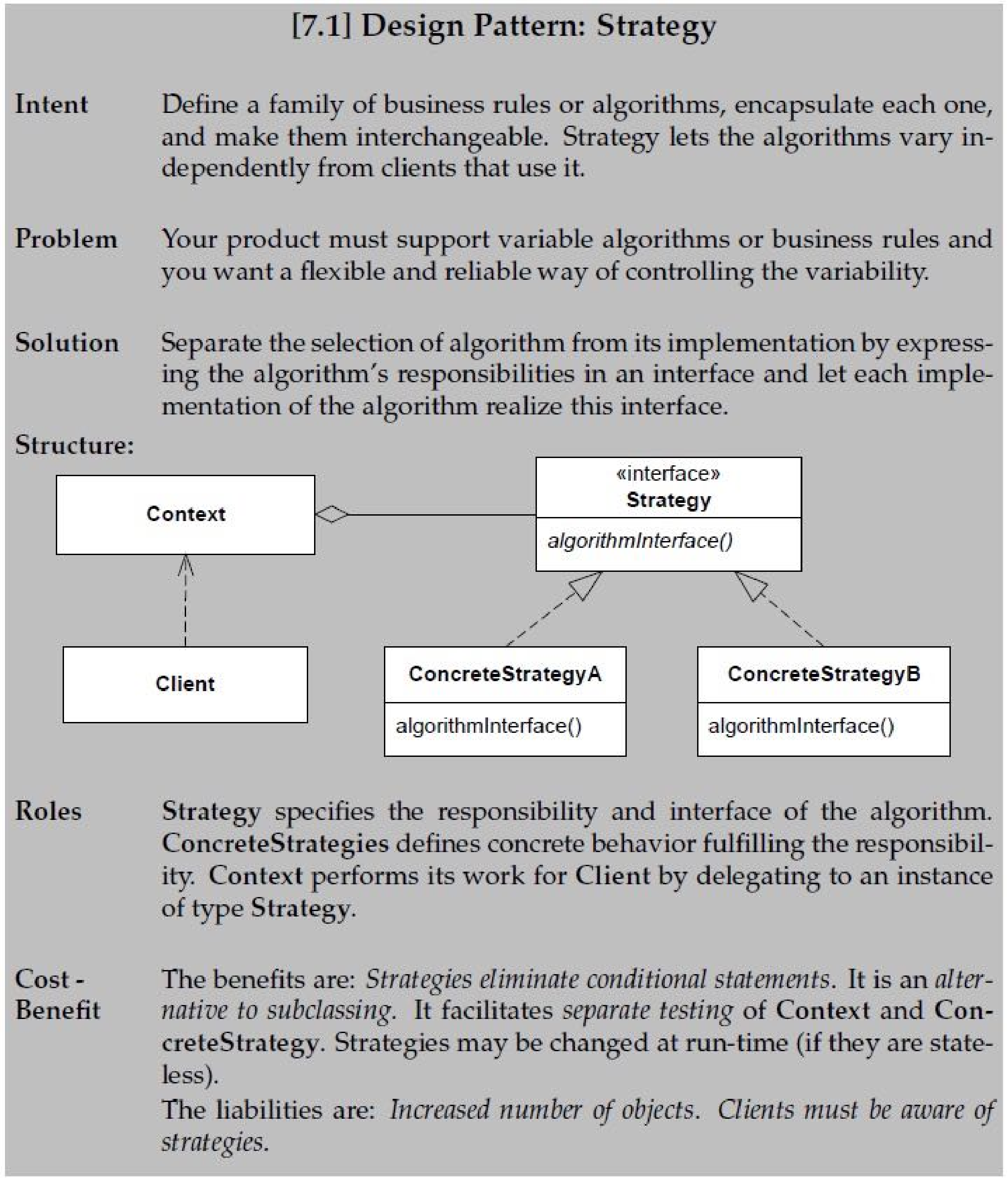
② **Delegate:** The parking machine now performs rate calculations by letting a delegate object do it: the RateStrategy object.

- time = rateStrategy.calculateTime(amount);

Design Patterns

Model 4: Composition is the **Strategy** design pattern!

- **Strategy** addresses the problem of encapsulating a family of algorithms / business rules and allows implementations to vary independently from the client that uses them



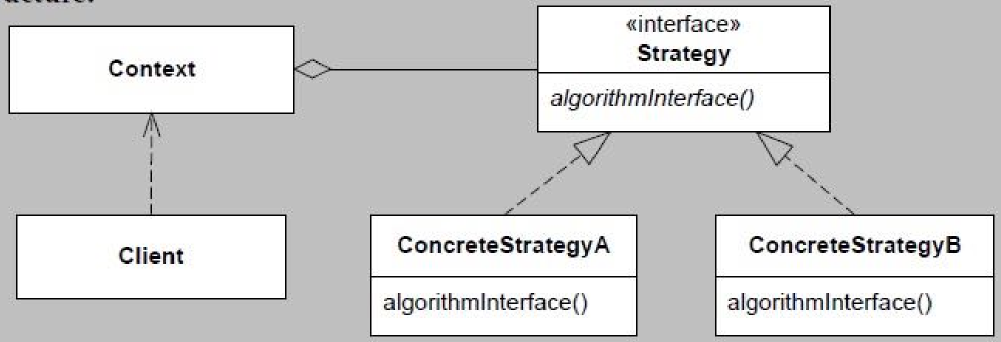
[7.1] Design Pattern: Strategy

Intent Define a family of business rules or algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms vary independently from clients that use it.

Problem Your product must support variable algorithms or business rules and you want a flexible and reliable way of controlling the variability.

Solution Separate the selection of algorithm from its implementation by expressing the algorithm's responsibilities in an interface and let each implementation of the algorithm realize this interface.

Structure:



Roles Strategy specifies the responsibility and interface of the algorithm. ConcreteStrategies defines concrete behavior fulfilling the responsibility. Context performs its work for Client by delegating to an instance of type Strategy.

Cost - Benefit The benefits are: *Strategies eliminate conditional statements*. It is an *alternative to subclassing*. It facilitates *separate testing* of Context and ConcreteStrategy. Strategies may be changed at run-time (if they are stateless).
The liabilities are: *Increased number of objects*. *Clients must be aware of strategies*.

Design Patterns

A **design pattern** is a solution to a problem in a context.

“Pattern” concept originally applied to houses, urban planning

→ Patterns at different scales, can be used recursively

The collection of patterns (**pattern catalog**) is a design tool.

Also a **communication** tool – “pattern language”.

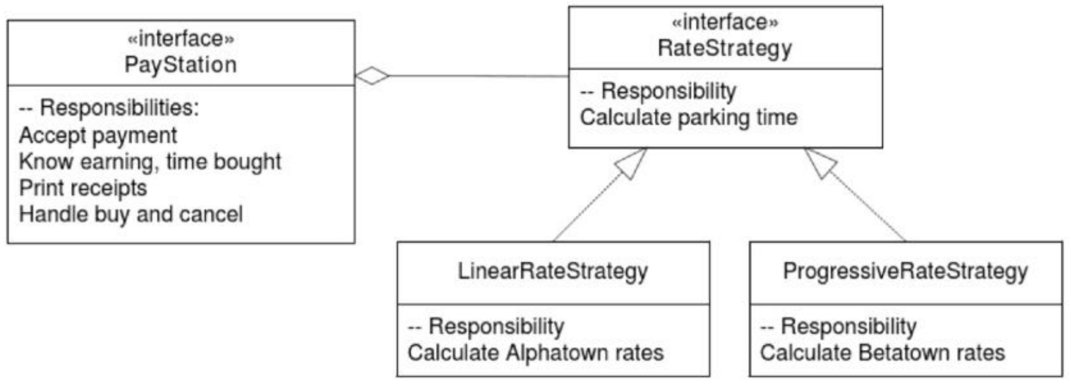
→ Facilitate design discussions

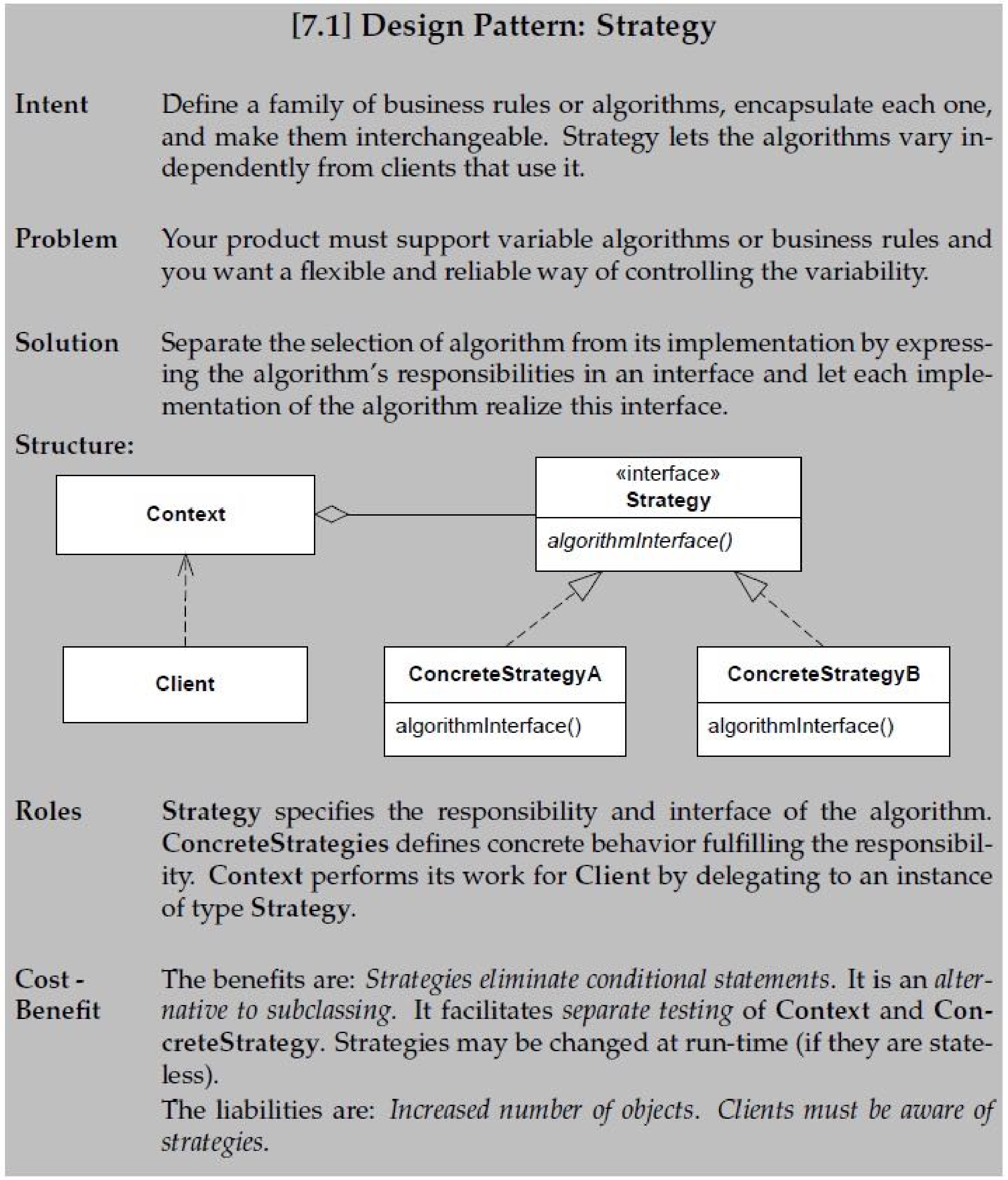
→ Represent knowledge gained through design experience

Next Steps

We need to **modify** the pay station software to implement the Strategy pattern.

- How can we **reliably** modify PayStationImpl?
- How do we ensure that we do not introduce **defects** during modification?





[7.1] Design Pattern: Strategy

Intent	Define a family of business rules or algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms vary independently from clients that use it.
Problem	Your product must support variable algorithms or business rules and you want a flexible and reliable way of controlling the variability.
Solution	Separate the selection of algorithm from its implementation by expressing the algorithm's responsibilities in an interface and let each implementation of the algorithm realize this interface.
Structure:	
Roles	**Strategy** specifies the responsibility and interface of the algorithm. **ConcreteStrategies** defines concrete behavior fulfilling the responsibility. **Context** performs its work for **Client** by delegating to an instance of type **Strategy**.
Cost - Benefit	The benefits are: *Strategies eliminate conditional statements*. It is an *alternative to subclassing*. It facilitates *separate testing* of **Context** and **ConcreteStrategy**. Strategies may be changed at run-time (if they are stateless). The liabilities are: *Increased number of objects*. *Clients must be aware of strategies*.

Refactoring

Stay focused, take **small steps** (recall TDD principles)

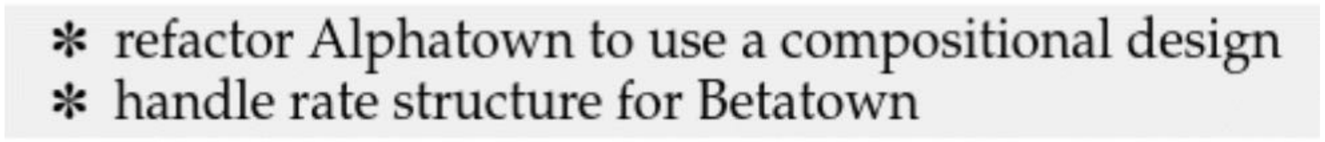
1. **Refactor** the current AlphaTown implementation to use compositional design (Strategy) using **existing test cases**
2. **Add** code to handle BetaTown's rate policy (in addition to preserving AlphaTown's)

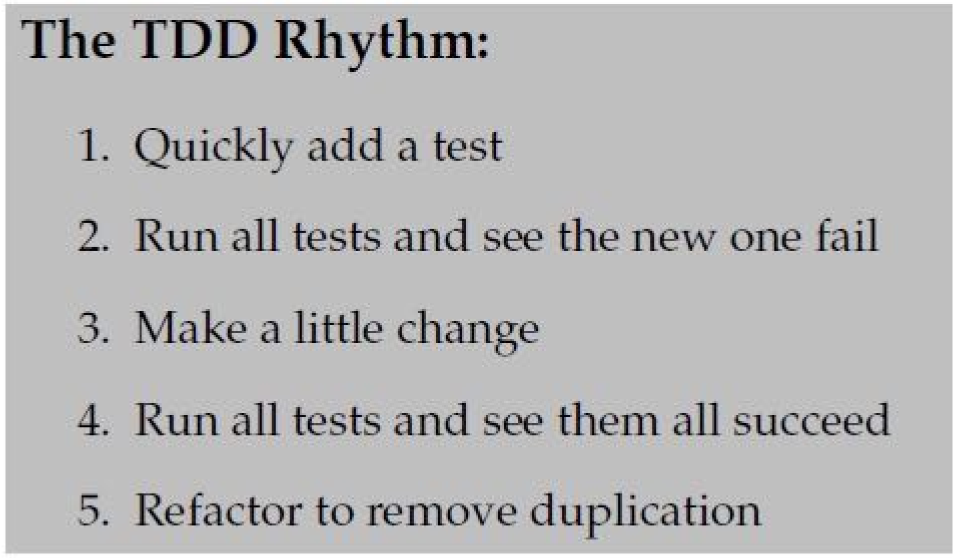
Refactoring is the process of changing a software system in such a way that **does not alter the external behavior** of the code yet **improves its internal structure**.

Fowler, 1999

Refactoring

Stay focused, take **small steps** (recall TDD principles)


* refactor Alphatown to use a compositional design
* handle rate structure for Betatown


The TDD Rhythm:

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

Difference when refactoring is that we start by modifying code instead of writing a new test.

But, we still use test cases as we make changes.

✻ refactor Alphatown to use a compositional design
✻ handle rate structure for Betatown

Refactoring

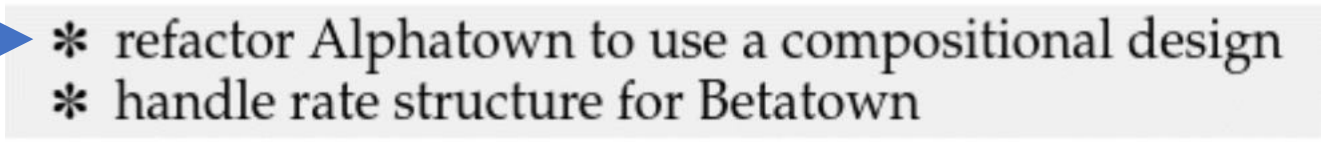
i. Introduce the RateStrategy interface

RateStrategy.java

```java
package paystation.domain;
/** The strategy for calculating parking rates.
*/
public interface RateStrategy {
  /**
   return the number of minutes parking time the provided
   payment is valid for.
   @param amount payment in some currency.
   @return number of minutes parking time.
   */
  public int calculateTime( int amount );
}
```

Lecture 7

Refactoring

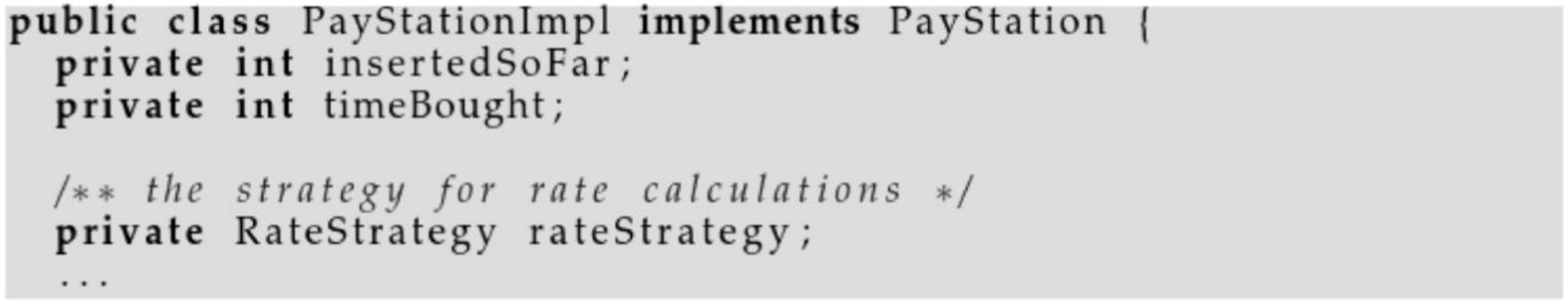


* refactor Alphatown to use a compositional design
* handle rate structure for Betatown

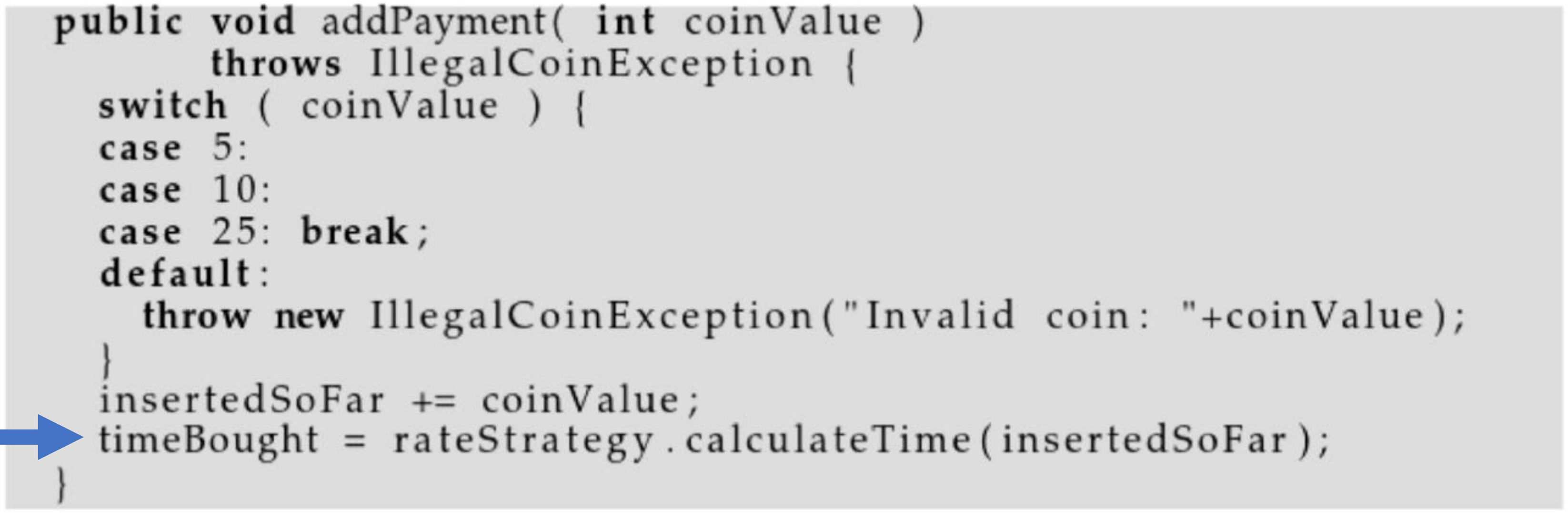
i. Introduce the RateStrategy interface (so that the next step compiles)

ii. Refactor PayStationImpl to use a reference to a RateStrategy instance for calculating rate (run tests to see them fail)

Delegate



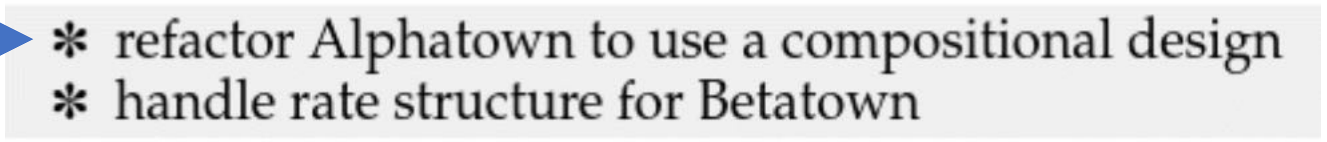
```
public class PayStationImpl implements PayStation {
  private int insertedSoFar;
  private int timeBought;

  /** the strategy for rate calculations */
  private RateStrategy rateStrategy;
  ...
```



```
 public void addPayment( int coinValue )
        throws IllegalCoinException {
   switch ( coinValue ) {
   case 5:
   case 10:
   case 25: break;
   default:
     throw new IllegalCoinException("Invalid coin: "+coinValue);
   }
   insertedSoFar += coinValue;
   timeBought = rateStrategy.calculateTime(insertedSoFar);
 }
```

Using existing test cases

→ Tests fail after this change

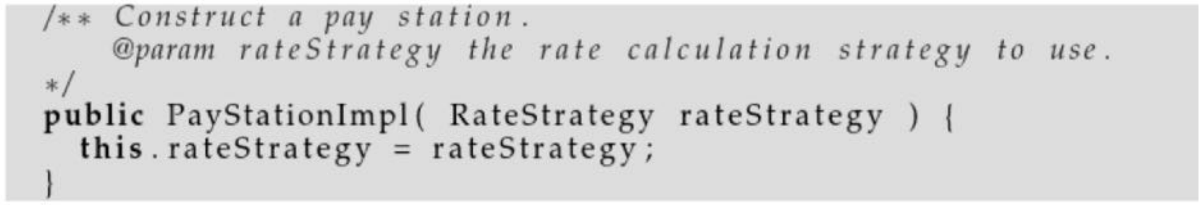


✻ refactor Alphatown to use a compositional design
✻ handle rate structure for Betatown

Refactoring

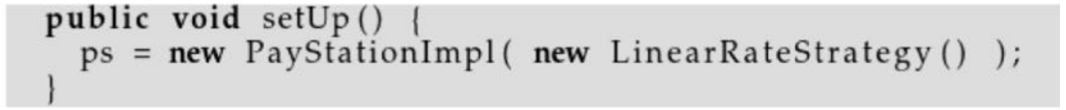
iii. Refactor PayStationImpl to use a concrete RateStrategy instance

PayStationImpl.java



```
/** Construct a pay station.
    @param rateStrategy the rate calculation strategy to use.
*/
public PayStationImpl( RateStrategy rateStrategy ) {
  this.rateStrategy = rateStrategy;
}
```

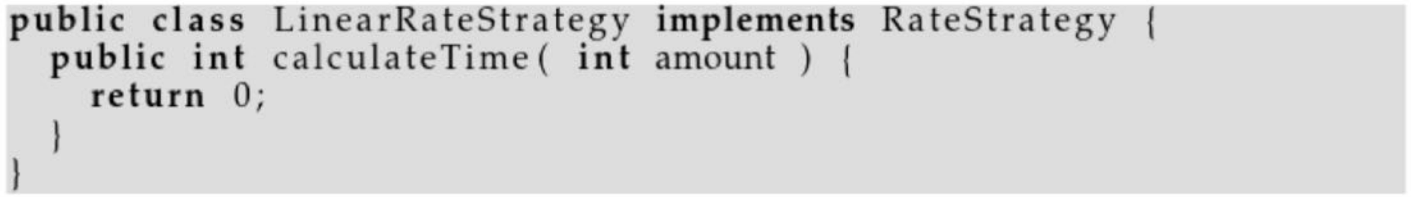
TestPayStation.java



```
public void setUp() {
  ps = new PayStationImpl( new LinearRateStrategy() );
}
```

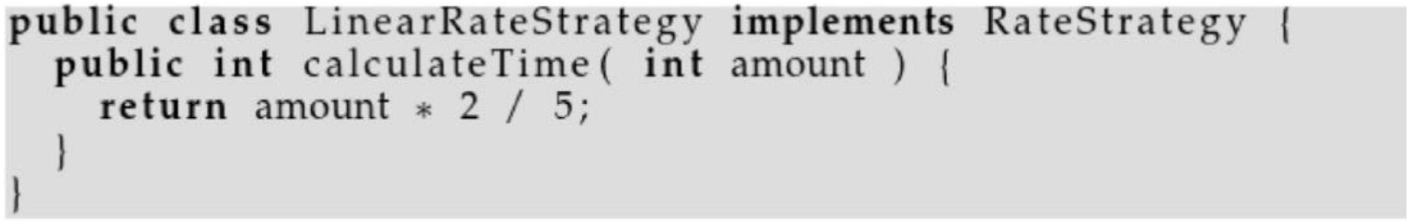
Also update test case setup

iv. Move rate calculation algorithm to a class implementing the RateStrategy interface



```
public class LinearRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return 0;
  }
}
```

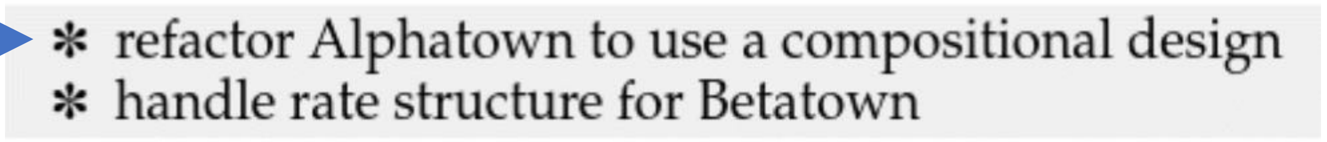
→ Tests fail by value



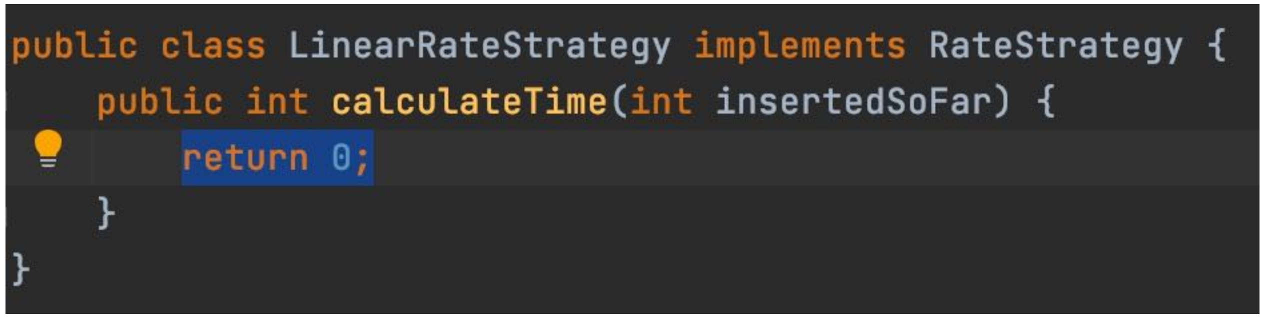
```
public class LinearRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return amount * 2 / 5;
  }
}
```

→ Tests pass
→ Refactoring complete!

Refactoring

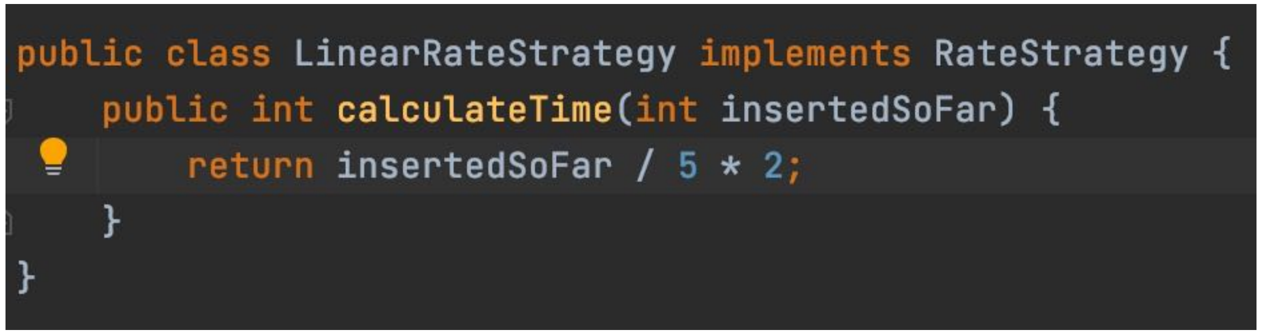


* refactor Alphatown to use a compositional design
* handle rate structure for Betatown



```
public class LinearRateStrategy implements RateStrategy {
    public int calculateTime(int insertedSoFar) {
        return 0;
    }
}
```

Solution-first programming: Statements that represent the solution are written first. They may refer to unknown interfaces, classes, methods, etc. – so use IDE suggestions to fill in the blanks.



```
public class LinearRateStrategy implements RateStrategy {
    public int calculateTime(int insertedSoFar) {
        return insertedSoFar / 5 * 2;
    }
}
```

Refactoring

Refactor the design **before** introducing new features

→ Make sure all existing tests pass!

Test cases should support refactoring

→ Refactoring is changing the implementation without changing external behavior

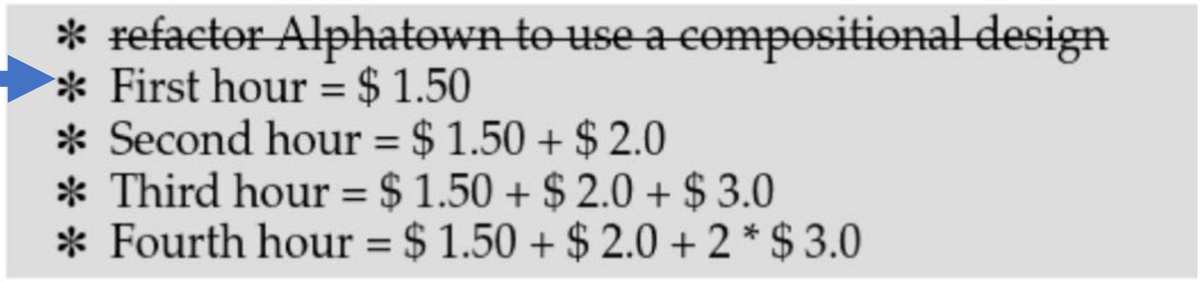
→ Test cases should not rely on implementation details

Do this: assertThat(game.getCityAt(p), is….)

Not this: assertThat(game.getInternalDataStruture().getAsArray()[47], is …)

TDD may seem like a nuisance when developing…

But, it ensures you have tests written to enable refactoring!



1 * ~~refactor Alphatown to use a compositional design~~
2 * First hour = $ 1.50
3 * Second hour = $ 1.50 + $ 2.0
4 * Third hour = $ 1.50 + $ 2.0 + $ 3.0
* Fourth hour = $ 1.50 + $ 2.0 + 2 * $ 3.0

BetaTown

To introduce the real BetaTown rate policy, we will use **Triangulation** (Abstract only when you have two or more examples)

Iteration **2:** Add test case for the first hour

Add just enough production code to make the test pass

Iteration **3**: Add test case for second hour

Add just enough complexity to the rate policy algorithm

Iteration **4:** Add test case for third and following hours

Add just enough more complexity

Unit and Integration Testing

We can actually test the new rate policy **independent of the Pay Station!**

TestProgressiveRate.java

```
@Before
public void setUp() {
  ps = new PayStationImpl( new ProgressiveRateStrategy() );
}

/** Test a single hour parking */
@Test public void shouldDisplay60MinFor150cent()
        throws IllegalCoinException {
  // First hour: $1.5
  ps.addPayment( 25 ); ps.addPayment( 25 );
  ps.addPayment( 25 ); ps.addPayment( 25 );

  ps.addPayment( 25 ); ps.addPayment( 25 );

  assertEquals( 60 /*minutes*/, ps.readDisplay() );
}
```

Unit testing is the process of executing a software unit in isolation to find defects within the unit itself

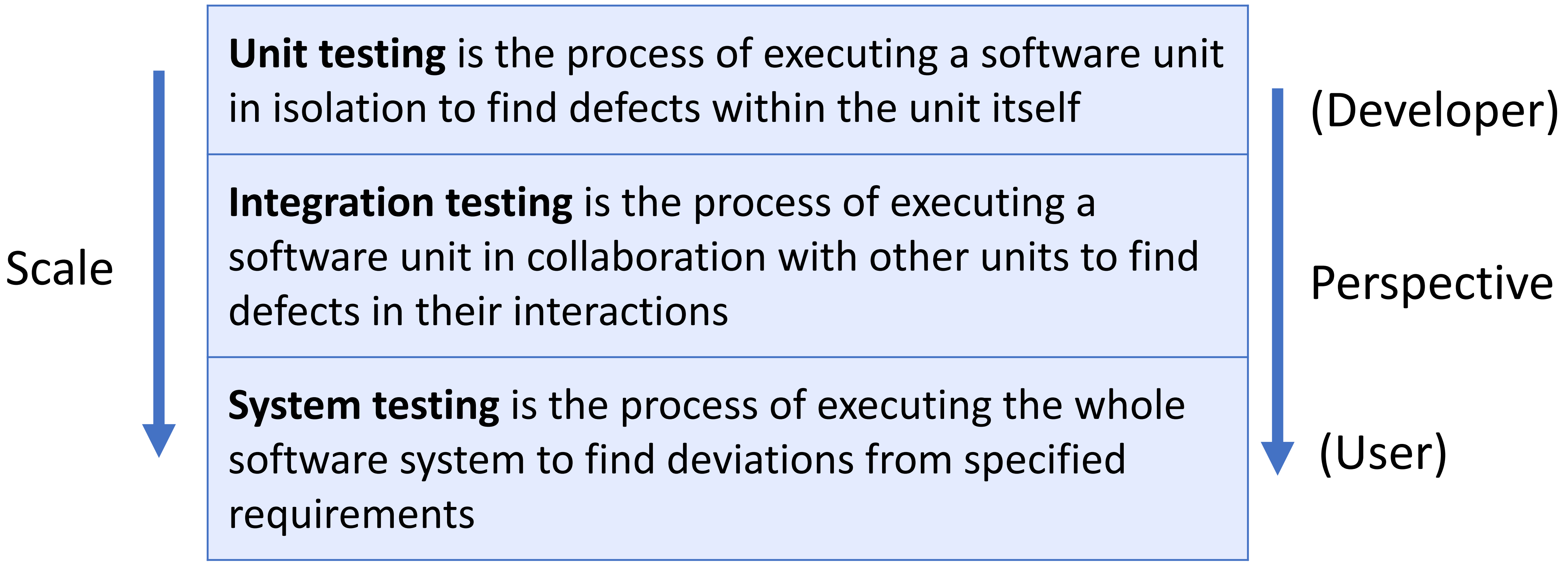
```
public class TestProgressiveRate {
  RateStrategy rs;

  @Before public void setUp() {
    rs = new ProgressiveRateStrategy();
  }
```

```
/** Test two hours parking */
@Test public void shouldDisplay120MinFor350cent()
        throws IllegalCoinException {
  // Two hours: $1.5+2.0
  addOneDollar();
  addOneDollar();
  addOneDollar();
  addHalfDollar();

  assertEquals( 2 * 60 /*minutes*/ , ps.readDisplay() );
}
```

```
@Test public void shouldGive120MinFor350cent() {
  // Two hours: $1.5+2.0
  assertEquals( 2 * 60 /*minutes*/ , rs.calculateTime(350) );
}
```

Unit and Integration Testing

Testing the parts does not mean that the whole is tested (and vice versa)!



Unit testing is the process of executing a software unit in isolation to find defects within the unit itself

Integration testing is the process of executing a software unit in collaboration with other units to find defects in their interactions

System testing is the process of executing the whole software system to find deviations from specified requirements

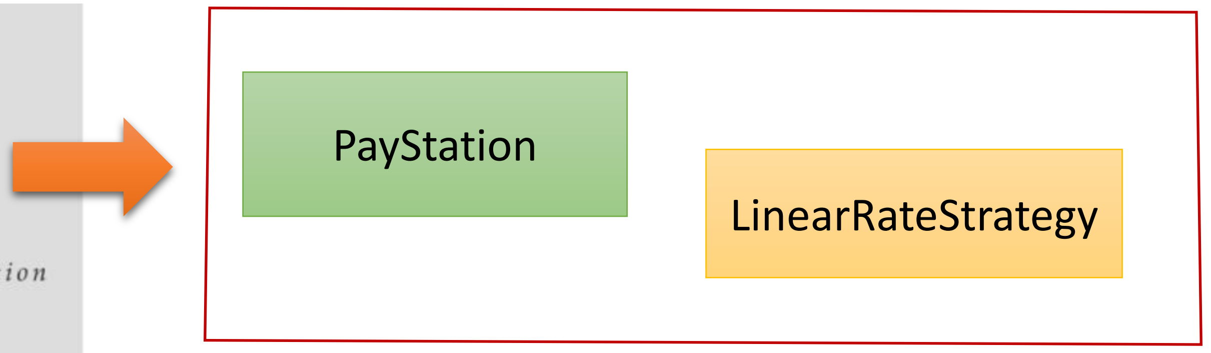
Defects can be caused by **interactions** between units with the wrong configuration!

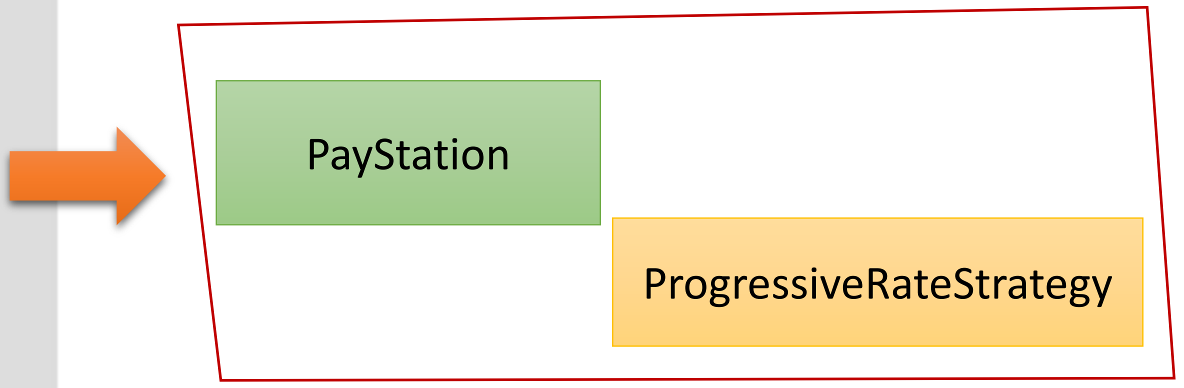
Unit and Integration Testing

Consolidate integration testing code:

TestIntegration.java

```
public class TestIntegration {
  private PayStation ps;

  /**
   * Integration testing for the linear rate configuration
   */
  @Test
  public void shouldIntegrateLinearRateCorrectly()
          throws IllegalCoinException {
    // Configure pay station to be the progressive rate pay station
    ps = new PayStationImpl( new LinearRateStrategy() );
    // add $ 2.0:
    addOneDollar(); addOneDollar();

    assertEquals( "Linear Rate: 2$ should give 80 min ",
                  80 , ps.readDisplay() );
  }
  /**
   * Integration testing for the progressive rate configuration
   */
  @Test
  public void shouldIntegrateProgressiveRateCorrectly()
          throws IllegalCoinException {
    // reconfigure ps to be the progressive rate pay station
    ps = new PayStationImpl( new ProgressiveRateStrategy() );
    // add $ 2.0: 1.5 gives 1 hours, next 0.5 gives 15 min
    addOneDollar(); addOneDollar();

    assertEquals( "Progressive Rate: 2$ should give 75 min ",
                  75 , ps.readDisplay() );
  }

  private void addOneDollar() throws IllegalCoinException {
    ps.addPayment(25); ps.addPayment(25);
    ps.addPayment(25); ps.addPayment(25);
  }
}
```





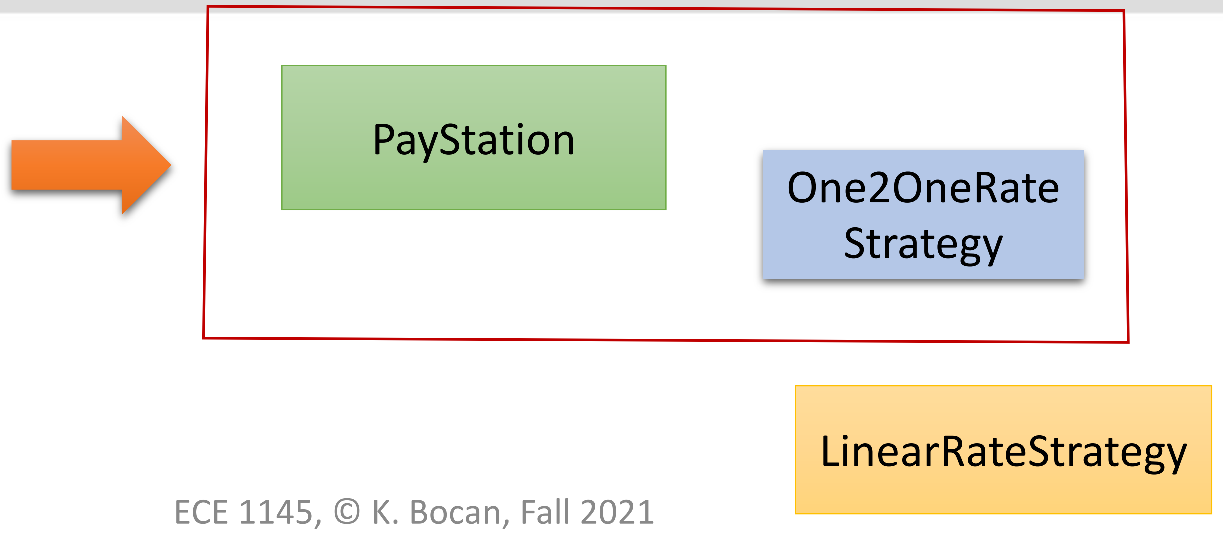
Unit and Integration Testing

How should we unit test the Pay Station? Which rate strategy should we use?

→ Introduce a very simple rate strategy for unit testing the pay station

src/**test**/java/paystation/domain/One2OneRateStrategy.java

```
package paystation.domain;
/** A simple one cent = one minute rate strategy for simplifying
    unit testing the pay station.
*/
public class One2OneRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return amount;
  }
}
```



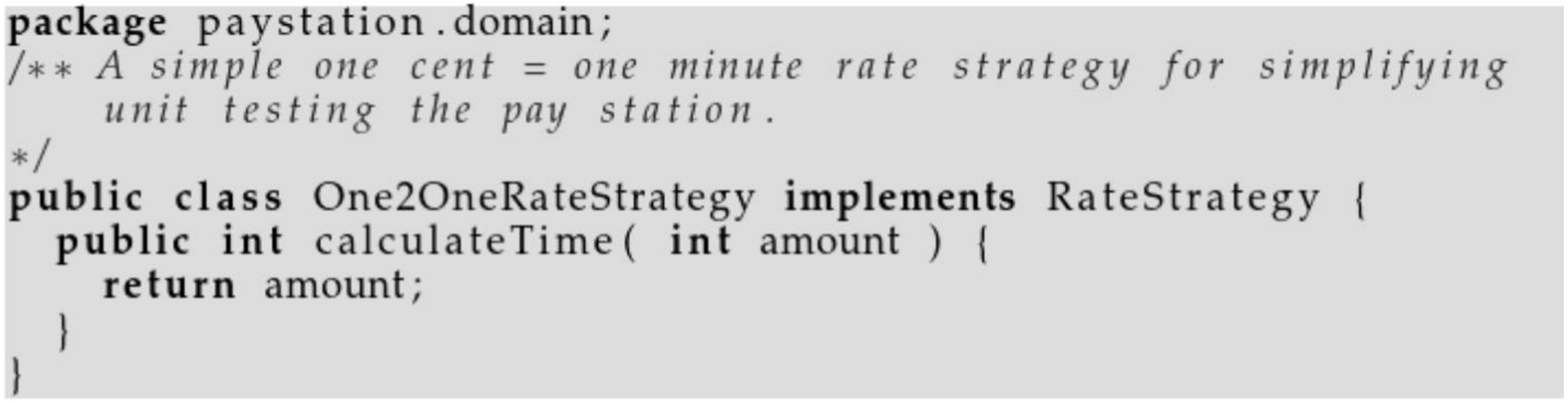
Unit and Integration Testing

How should we unit test the Pay Station? Which rate strategy should we use?

→ Introduce a very simple rate strategy for unit testing the pay station

Only in test code

src/**test**/java/paystation/domain/One2OneRateStrategy.java



```
package paystation.domain;
/** A simple one cent = one minute rate strategy for simplifying
    unit testing the pay station.
*/
public class One2OneRateStrategy implements RateStrategy {
  public int calculateTime( int amount ) {
    return amount;
  }
}
```

Keep all testing related code in the test tree!

→ If a class is not used in the production code

Integration vs. System Testing

Integration testing is not system testing!

System testing is testing the **full system** ("Functional Testing" in XP):
Test that A works with **real** B, **real** C, **real** D, and **real** E units
(e.g., databases, servers, hardware, etc.)

We typically integration test with "stubs" representing real units – more later.

→ Use cases drive system testing (collaborate with customer)

While unit/integration tests must pass, **system tests** may not pass at 100% until project completion.

Why do we code?

Code is written to:

- Be compiled, deployed, and executed by users in order to serve their need
- **Be maintained – that is, read and understood by humans so it can easily and correctly modified**

Functionally correct code can be next to impossible to maintain if it is not well-structured / well-written / well-documented (/well-understood)…

Clean Code

- Small
  - Make functions/methods do 1-5 logical steps
- Do One Thing
  - Do one thing, do it well, do it only! Keep focus!
- One Level of Abstraction
  - Don't break encapsulation, hide implementation details
- Use Descriptive Names
  - Describe the one thing it does! Do not describe anything else
- Keep Number of Arguments low
  - 0-1-2 maybe three arguments.
- Avoid Flag Arguments
  - Indication that a method is doing **more than one thing**
- Have No Side Effects
  - Do not do hidden things / hidden state changes
- Command Query Separation
  - Query: no state change, return a value
  - Command: no return value, change the state
- Prefer Exceptions to Error Codes
  - Unless you have modules that cannot propagate exceptions (e.g., networking)
- Don't Repeat Yourself
  - Avoid **multiple maintenance problem**
- No arguments in method/class names
  - A symptom of duplicated code
  - Exception: test case methods
- Do the same thing the same way
  - Analyzability
- Name Boolean sub-expressions
  - Analyzability
- Avoid lots of nesting
  - Flatten by bailing out as soon as an answer can be computed

Make it **work**, then make it clean.

Lecture 8

Refactoring: Improving the Design of Existing Code

Code Smells

Martin Fowler, Kent Beck

“Code smell” is a term for a surface indication that usually corresponds to a deeper problem in the system (implied that smell is bad).

- Find areas of the code that can be improved
- Identify where refactoring is needed
- Address problems **early** before they have larger effects

Presence of a ”smell” does not always mean there is a problem!

Code Smells

Martin Fowler, Kent Beck

- Mysterious Name
- Duplicated Code
- Long Function
- Long Parameter List
- Global Data
- Mutable Data
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Data Clumps
- Primitive Obsession
- Repeated Switches
- Loops
- Lazy Element
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Insider Trading
- Large Class
- Alternative Classes with Different Interfaces
- Data Class
- Refused Bequest
- Comments

Code Smells

Martin Fowler, Kent Beck

- **Global Data**
  - Encapsulate variables (in a function / class)
- **Mutable Data**
  - Encapsulate data such that updates occur through functions
- **Data Clumps**
  - If bunches of data tend to always show up together, consider making a class
- **Insider Trading**
  - Minimize passing data, and make it obvious when needed
- **Data Class**
  - Move more behavior from classes that are setting/getting into the data class

Code Smells

Martin Fowler, Kent Beck

- **Divergent Change**
  - Fix by separating contexts
- **Shotgun Surgery**
  - Move code such that changes occur in a single module
- **Speculative Granularity**
  - Make changes as needed, not in anticipation of need
- **Alternative Classes with Different Interfaces**
  - Use interfaces for consistent interaction with a group of objects
- **Comments**
  - Don’t use comments to compensate for unreadable code; why, not how

Cohesion and Coupling

Recall:

Coupling is a measure of how strongly dependent one software unit is on other units

Cohesion is ta measure of how strongly related and focused the responsibilities and behaviors of a software unit are

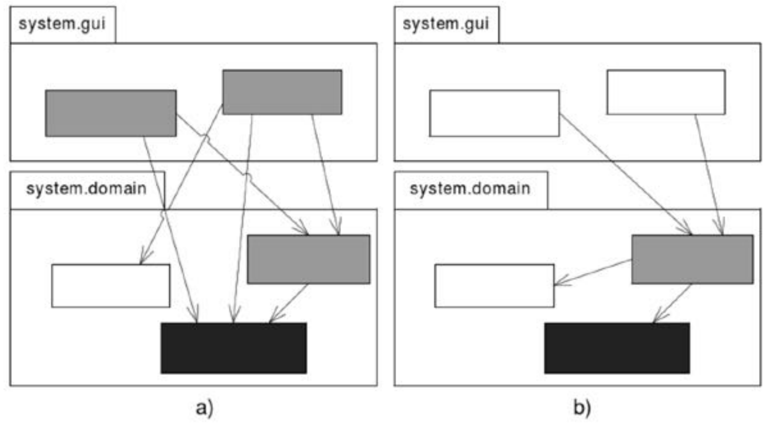


Figure 10.1: Tight (a) and low (b) coupling.

Assign responsibility so coupling is low, and cohesion is high

→ **Lower the cost of change**

Quality of Test Code

TDD produces two code bases

- The **production** code
- The **test** code

Code quality is important in both!

- You want to be able to read and understand test code
- **Test code exception to "clean code": Arguments in Argument Lists**
  - "moveRedArcherToPosition45()"

Summary

Code quality guidelines are “rules of thumb” for analyzability

Remember: Actual “rules” vary, depend on:

- Company culture
- Preferences
- Most important: Agreement among developers

What is code review?

Let other people read your code (and vice versa) and provide suggestions or make changes.

Why code review?

→ Build a shared understanding of the code base / reach consensus on changes

Benefits of code review:

- Knowledge transfer, establishing best practices
- Improving code clarity
- Identifying refactoring opportunities
- Sharing perspectives/generating new ideas

Sometimes refactoring is done during the code review

What to look for?

- Design
- Functionality
- Complexity
- Tests
- Naming
- Comments
- Style
- Consistency
- Documentation
- Every Line
- Context
- Good Things

There is no such thing as perfect code, only **better** code.

Guidelines

SMART Code Review:

Be **S**pecific with defects

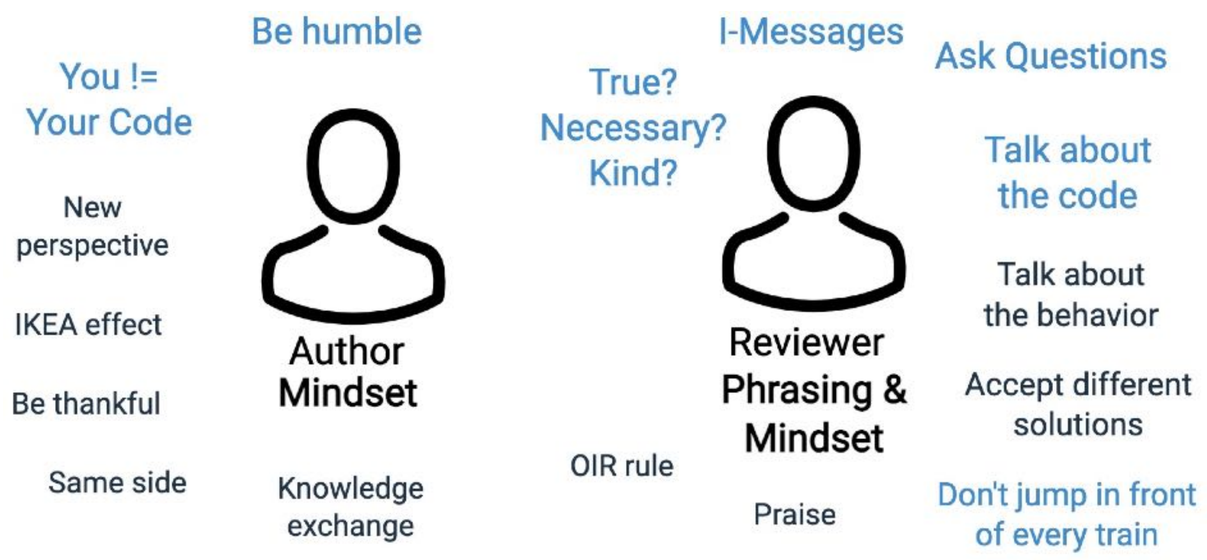
Have **M**easurable and **A**chievable suggested improvements

Give feedback that is **R**elevant to the type of review and the problem to be solved

Give feedback that is **T**ime bounded – set priorities and have specific actions to be taken to remove defects with a specified deadline

Implementing Effective Code Reviews, Giuliana Carullo

Guidelines



Code review guidelines for doing code reviews like a human.

Before giving feedback, ask yourself:

- Is it true?
  - Express your **opinion**, use I-statements
- Is it necessary?
  - Focus on the most important changes, be **helpful**
- Is it kind?
  - Provide useful criticism and feedback without shaming the author