

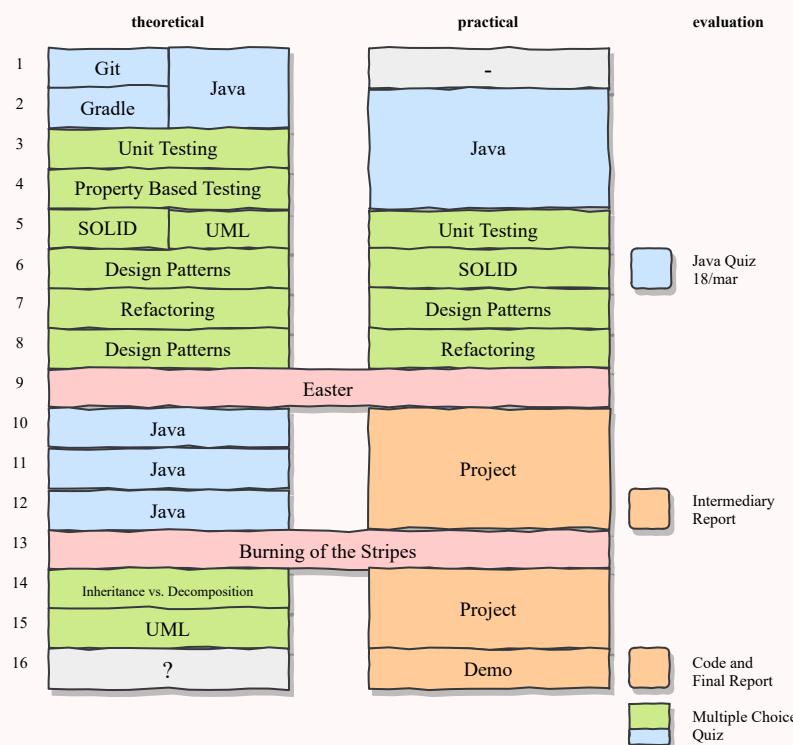
LPOO

André Restivo

Course Content

- Git / Java / Gradle
- Unit Testing
- SOLID Principles
- UML: Class and Sequence diagrams
- Design Patterns
- Refactoring and Code Smells

Master Plan

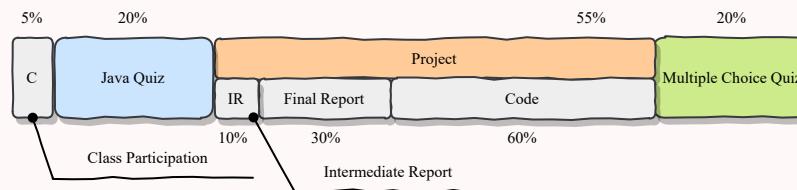


Main Bibliography

- Bruce Eckel; Thinking in Java. ISBN: 0-13-027363-5 (4th edition)
- Russ Miles and Kim Hamilton; Learning UML 2.0. ISBN: 978-0-596-00982-3
- Kent Beck; Test-driven development. ISBN: 978-0-32-114653-3
- Erich Gamma... [et al.]; **Design Patterns**. ISBN: 0-201-63361-2
- Martin Fowler ; with contributions by Kent Beck... [et al.]; **Refactoring**. ISBN: 0-201-48567-2

Evaluation

- To obtain frequency, students may not exceed the maximum number allowed of missed classes. Attendance will be registered in practice sessions.
- You must obtain a minimum of **40%** in all evaluation components.
- Final grade will be calculated as follows:



Git

André Restivo

Index

Introduction

Git Basics

Local

Branches

Remotes

Reverting

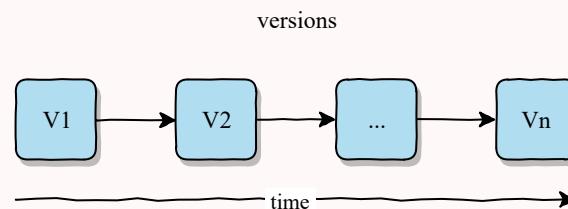
Workflows

More

Introduction

Version Control Systems (VCS)

A system that **records changes** to a file or set of files **over time**.



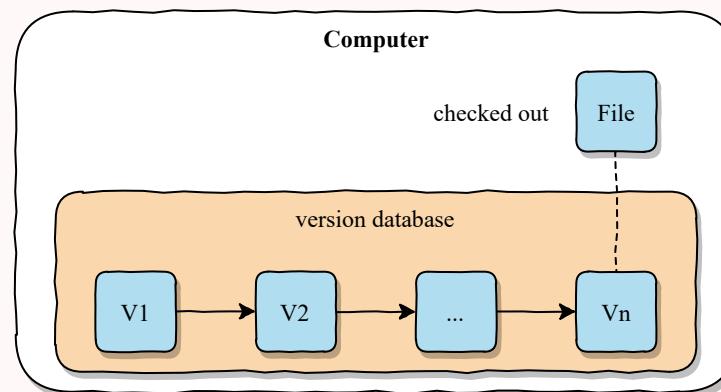
It allows you to:

- **revert** selected files, or a project, back to a previous state
- **compare** changes over time
- see **who modified** something
- ...

AKA Source Control Management (SCM)

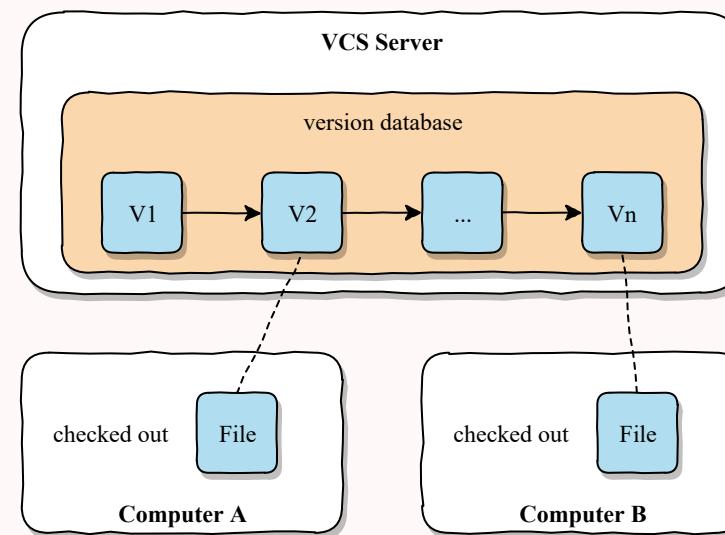
Local VCS

- Local VCS use a **simple database** that keeps all changes to files under revision control.
- Most store only the **differences** between files instead of copies of each version.
- Examples: **RCS**



Centralized VCS

- A **single server** that contains all the versioned files.
- Users can **checkout** a particular file version.
- Examples: **CVS, Subversion**



Centralized VCS

Advantages:

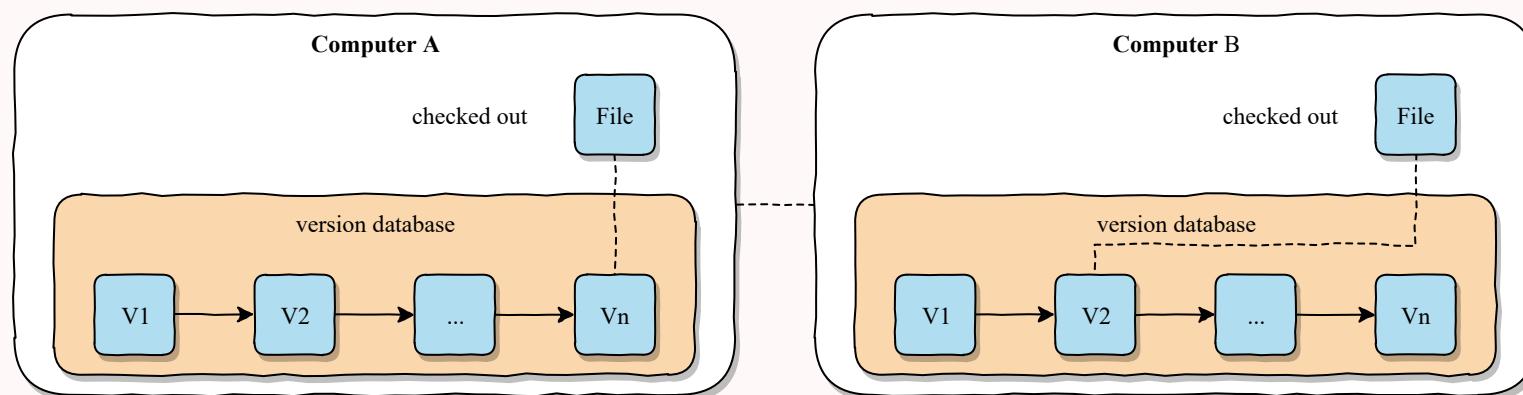
- Everyone knows what everyone is doing.
- **Fine grained control** over who can do what.

Disadvantages:

- Single point of **failure**.
- Needs constant **connectivity**.
- Backups are **mandatory**.

Distributed VCS

- All clients fully mirror the repository, including its **full history**.
- There is **no difference** between a server and a client.
- But one, or more, computers can be used as a **central point** of synchronization.
- Allows lots of different **workflows**.



Examples: [Git](#), [Mercurial](#), [Bazaar](#), [Darcs](#)

Git Basics

Basics

Snapshots:

- Does **not** store only the differences between versions of a file.
- Instead, it saves them as a series of **snapshots**.
- But, if files have not changed, it does not store them again ([link](#)).

Local: Most Git operations are **local**.

Integrity:

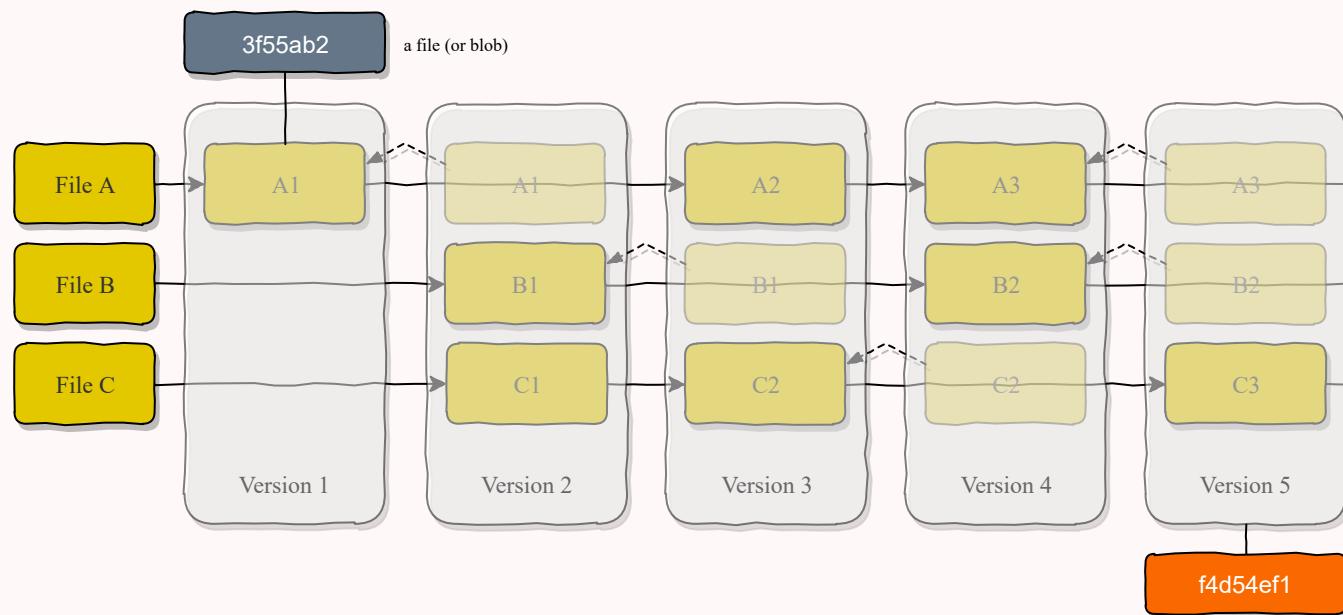
- Everything in Git is *checksummed* ([SHA-1](#)) before it is stored.
- Everything is then **referred** to by that **checksum**.
- Checksum example: 7e16b5527c77ea58bac36ddda6f5b444f32e81b

Versions

Each version (aka a **commit**) is a snapshot of that version files.

If not changed, files are just **links** to a previous version.

All objects (files, commits, ...) have an **hash identifier**.

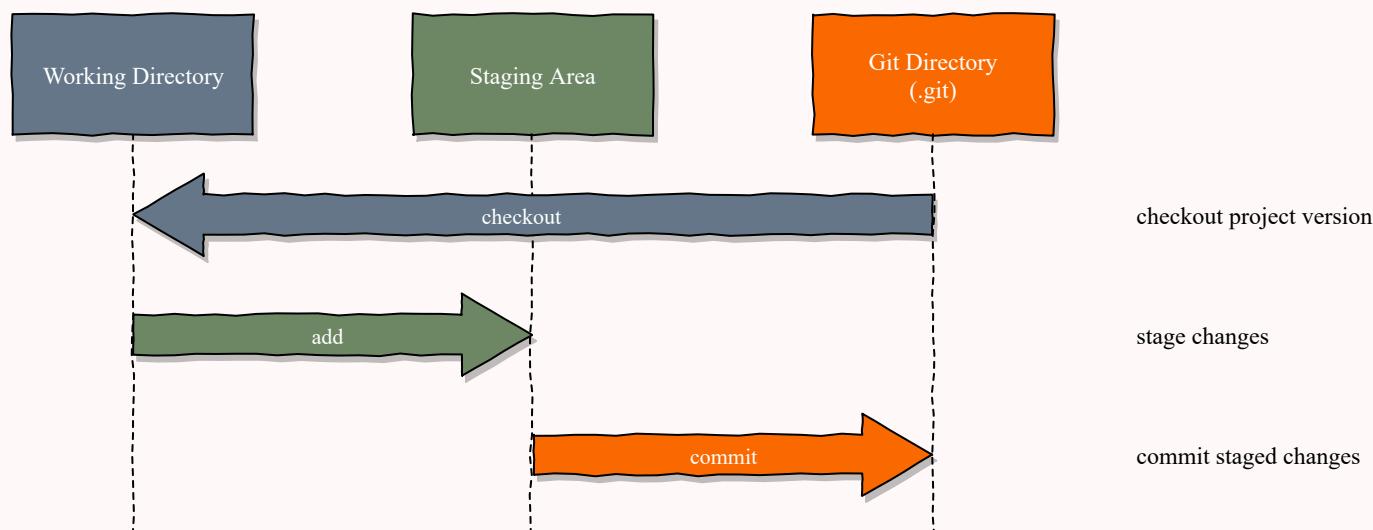


Git Areas

The **Git directory** (.git) is where Git stores the metadata and object database for your project.

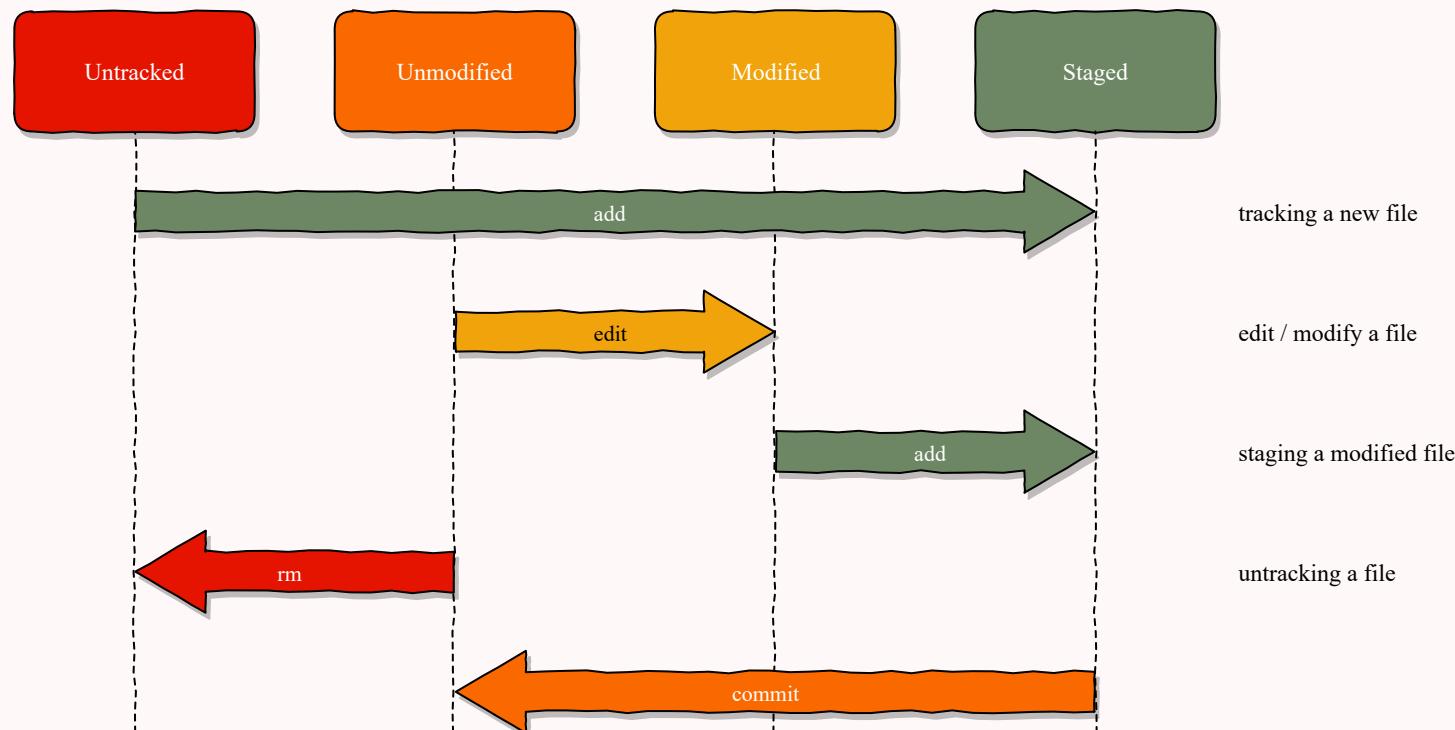
The **working tree** is a single **checkout** of one version of the project.

The **staging area (or index)** is a file in your Git directory that stores information about what will go into your next commit.



File States

Files in the working directory can be in different states:



Local Git

Git as a local VCS

Create a Repository

Enter a local directory, currently not under version control:

```
cd project
```

And turn it into a Git repository:

```
git init
```

This will create an hidden `.git` subdirectory containing all of your necessary repository files.

Add

The **add** command can be used to:

1. Track and stage a file that is currently **not tracked** by Git.
2. Stage a file that has been **modified**.

```
$ echo "hello git" > README      # File is created  
$ git add README                  # File is now tracked and staged
```

You can use the **--all** or **-A** flag to stage all untracked or modified files.

```
$ echo "hello git" > README      # File is created  
$ git add --all                   # File is now tracked and staged
```

Commit

The **commit** command records a new snapshot to the repository:

```
$ echo "hello git" > README      # File is created  
$ git add README                 # File is now tracked and staged  
$ git commit                      # Commits the file
```

After running commit, Git will open your **predefined** text editor so that you can write a small commit message (or use the **--message** or **-m** flag).

The **--all** or **-a** flag automatically stages any **modified** (tracked) files:

```
$ echo "goodbye git" > README      # Already tracked file is modified  
$ git commit -a -m "Edited README" # Stages and commits the file
```

Status

The `status` command can be used to determine which files are in which state:

```
$ echo "hello git" > README      # File is created
$ git status                      # Asking for file status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

nothing added to commit but untracked files present (use "git add" to track)
```

The `--short` (or `-s`) flag can be used to get a more concise output:

```
$ git status --short          # Asking for file status
?? README
$ git add README              # File is now tracked and staged
$ git status --short          # Asking for file status
A  README
```

Status

Notice that the `git status -s` command consists of two columns for each file.

```
$ echo "hello git" > README      # File is created  
$ git status -s  
?? README                         # File is untracked
```

The first column has information about the **staging area** and the second one about the **working directory**. In this case the file is untracked on both.

```
$ git add README                  # Modifications are staged  
$ git status -s  
A README                          # File added to staging area
```

Now the file has been **added** in the staging area.

```
$ git commit -m "Added README"    # Committing changes  
$ git status -s
```

Now the file has been **committed** and is **unmodified**.

Partially Staged Files

A file can be partially staged:

```
$ echo "some text" > README      # File is modified
$ git add README                  # Modifications are staged
$ echo "another text" >> README # File is modified again
$ git status -s
AM README                         # Added to staging area and modified
```

1) Committing again would **only** commit the initial staged edits:

```
$ git commit -m "Added some text"      # Committing initial edit
$ git status -s
M README                           # File now still has changes
$ git add README                    # Staging those changes
M README
$ git commit -m "Added another text" # Committing following edits
```

2) We could also **add** the new modifications first and **only commit** once:

```
$ git add README                  # Staging following changes
$ git status -s
A README                          # All changes staged
$ git commit -m "Added some and another text" # Committing both changes at once
```

Remove

If you delete a file from your working area, it will appear as a change that needs to be staged in order to be reflected in the repository:

```
$ rm README          # File is removed from working directory  
$ git status -s  
D README           # File removed in working tree  
$ git add README    # File removal is staged  
$ git status -s  
D README           # File removed in staging area  
$ git commit -m "Removed README" # File removal is committed
```

The **git rm** command simplifies this operation by removing the file from the working directory and staging that change at the same time.

```
$ git rm README      # Removed from working directory and staged  
$ git status -s  
D README           # File removed in staging area  
$ git commit -m "Removed README" # File removal is committed
```

History

The **log** command allows you to see the **commit history** of a repository.

```
$ git log
commit 41138ac70c5b32239c0000824d8d64315cb50d84 (HEAD -> master)
Author: User <user@email.com>
Date:   Thu Feb 7 09:55:36 2019 +0000

    Modified README

commit 5621668b7f21c4a06385e123d6ee20d1beb6fa1d
Author: User <user@email.com>
Date:   Thu Feb 7 09:55:15 2019 +0000

    Added README
```

- We can see by **whom** and **when** each commit was made.
- We can see the commit **message**.
- And also the **hash** of each commit.

Simplified History

The `--oneline` flag produces a simplified version of the log.

```
$ git log --oneline  
41138ac (HEAD -> master) Modified README  
5621668 Added README
```

We can also limit the number of entries to be shown.

```
$ git log --oneline -1  
41138ac (HEAD -> master) Modified README
```

Patches

The **--patch** (or **-p**) flag shows the difference (the **patch** output) introduced in each commit.

```
$ git log -1 -p
commit 41138ac70c5b32239c0000824d8d64315cb50d84 (HEAD -> master)
Author: User <user@email.com>
Date:   Thu Feb 7 09:55:36 2019 +0000

    Modified README

diff --git a/README b/README
index 7b57bd2..2e24352 100644
--- a/README
+++ b/README
@@ -1 +1,2 @@
    some text
+another text
```

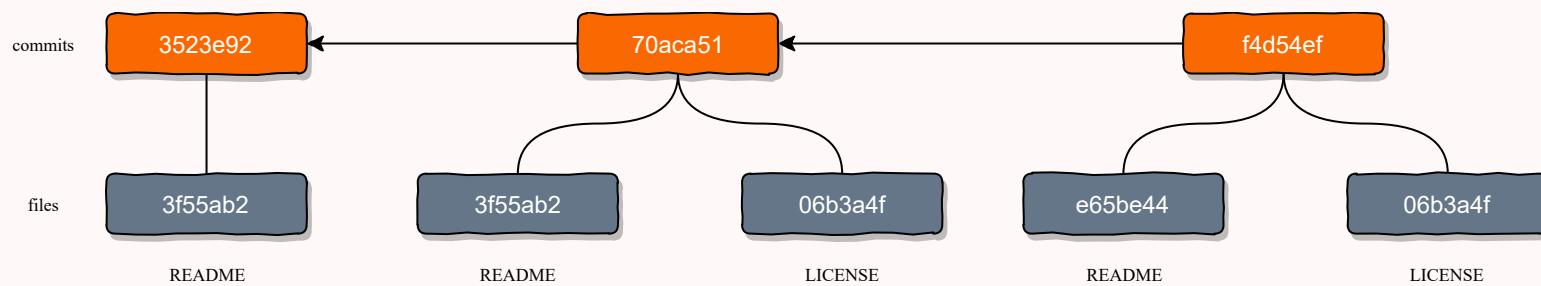
The output is rather intimidating but it allows you to see what changed in each commit.

Branches

Commits

As we have seen before, files are stored as **blobs** and identified by an **hash**.

Versions (or commits) are just a **snapshot**, also identified by an **hash**, pointing to a series of blobs.

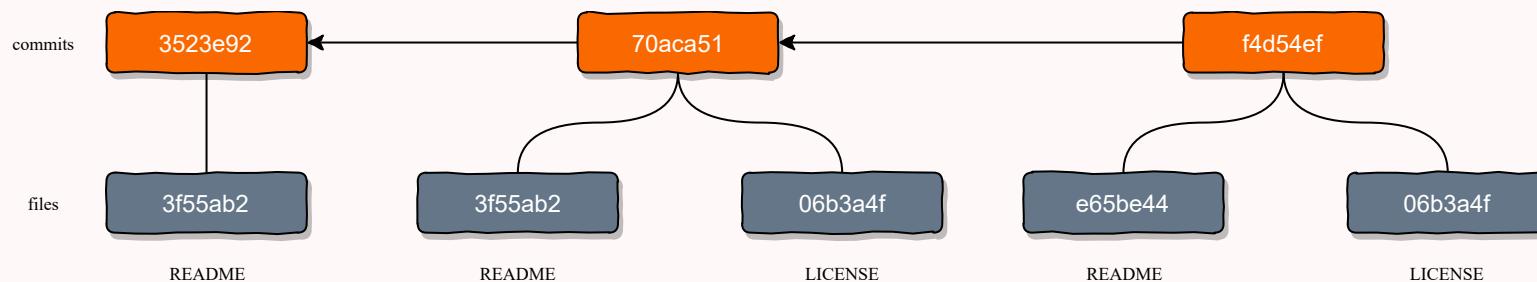


Each commit contains the author's **name** and **email** address, the **message** that was typed, and pointers to the commit (or commits) that directly came before this commit (its **parent** or **parents**).

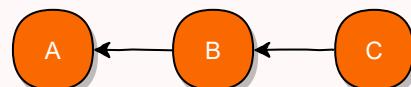
Commits

In this specific example we have 3 commits:

1. **3523e920** - The initial commit where a README file was added.
2. **70aca513** - A second commit where a LICENSE file was added.
3. **f4d54ef1** - A third commit where the README file was modified.



From now on, we will use a simplified version of this commit tree:

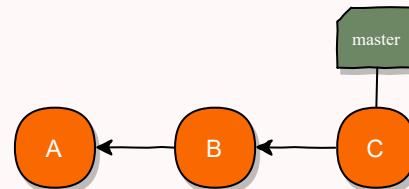


Branches

A **branch** in Git is simply a lightweight movable **pointer** to one of these commits.

The **default** branch name in Git is *master*.

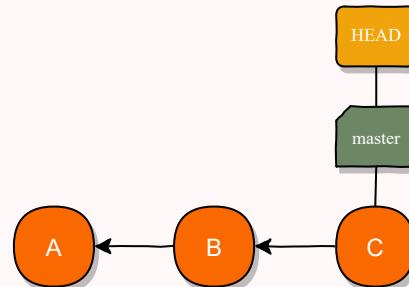
As you start making commits, you're given a *master* branch that points to the **last** commit you made.



Every time you commit, the *current* branch pointer moves **forward automatically**.

Head

Git uses a special pointer called **HEAD** that always points to your **current branch**.



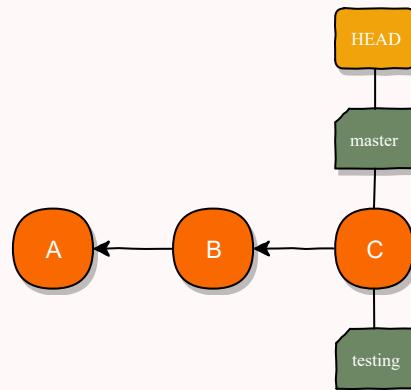
And now this makes a little bit more sense:

```
$ git log --oneline
f4d54ef (HEAD -> master) Modified README
70aca51 Added LICENSE
3523e92 Added README
```

Creating Branches

To create a branch we use the **branch** command. This only creates the branch, it does not move the HEAD:

```
$ git branch testing
```



The **branch** command can also show the current local branches.

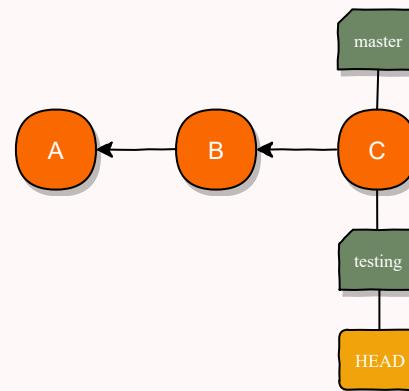
```
$ git branch  
* master  
  testing
```

The asterisk (*) represents the HEAD

Checkout

To **change** to another branch we can use the **checkout** command:

```
$ git checkout testing
  master
* testing
```



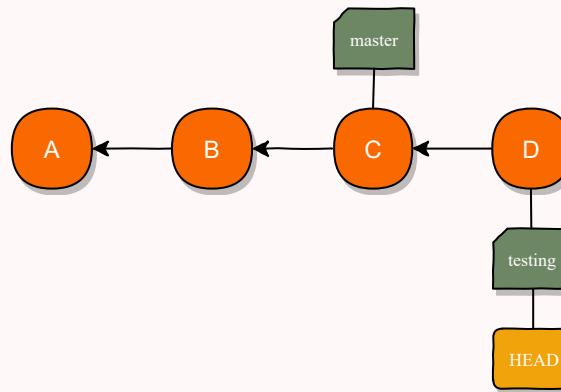
We can also **create and checkout** a new branch using the **-b** flag:

```
$ git checkout -b testing
```

Moving the HEAD

If we create a new commit now:

```
$ echo "more license info" >> LICENSE
$ git commit -a -m "Testing LICENSE"
```



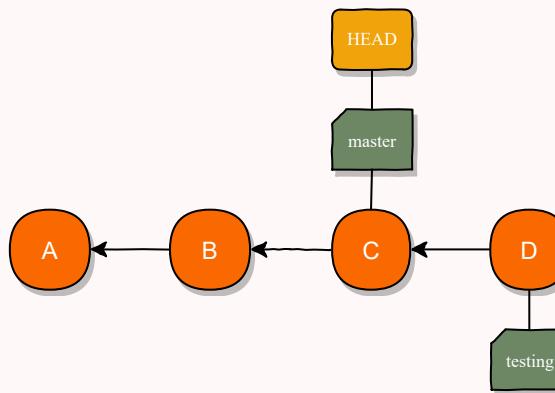
We can see that **only the current branch**, the one pointed by the HEAD, moved.

Checkout

If we **checkout** the *master* branch again, two things happen:

```
$ git checkout master
```

1. The HEAD moves to the commit pointed by the **master** branch.
2. Our files are reverted to the snapshot that **master** points to.

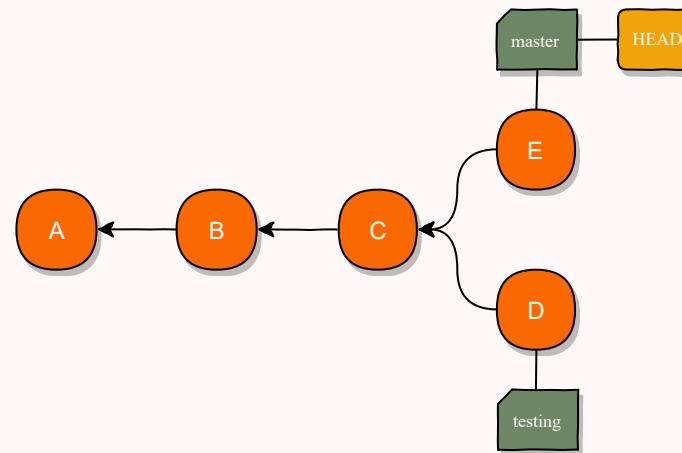


This means we are now working on top of a version that has **already been changed**. Any changes we make will create a **divergent history**.

Divergent Histories

Now that we are back to our master branch, let's do some more changes:

```
$ git checkout master
$ echo "license looks better this way" >> LICENSE
git commit -a -m "Better LICENSE"
```



Now we have two divergent histories that have to be **merged** together.

Merging

Merging is done by using the **merge** command:

```
$ git checkout master  
$ git merge testing
```

Git merges the **identified** branch into the **current** branch.

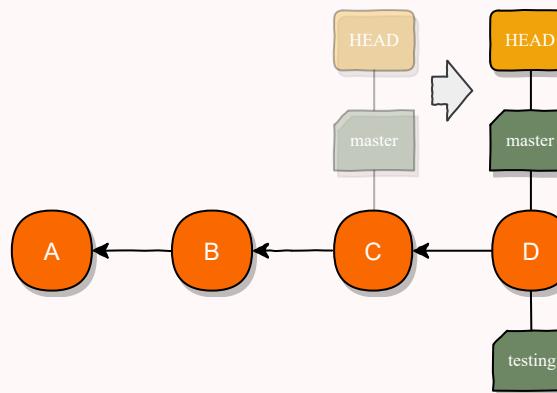
Git uses **two main strategies** to merge branches:

- Fast-forward merge: when there is **no divergent** work
- Three-way merge: when there is **divergent** work

Fast-forward Merge

When you merge one commit with a commit that can be reached by following the first commit's history because there is **no divergent work** to merge together, Git just **moves the branch pointer forward**.

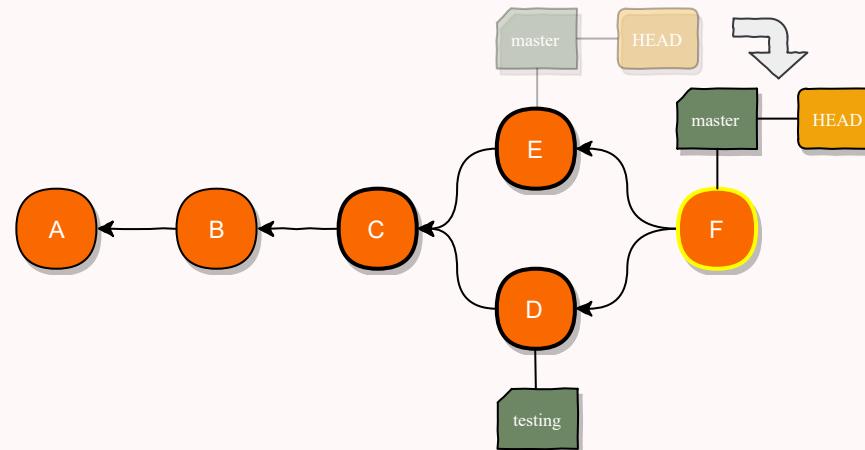
```
$ git checkout master  
$ git merge testing
```



Three-way Merge

When the commit on the branch you're on **isn't a direct ancestor** of the branch you're merging in, Git uses the **two snapshots** pointed to by the **branch tips** and the **common ancestor** of the two to create **a new commit**.

```
$ git checkout master  
$ git merge testing
```



Deleting Branches

If you do not need a branch any longer, you can just **delete** it.

Deleting a branch leaves all commits alone and only **deletes the pointer**.

```
$ git branch -d testing
```

Conflicts

If you changed the **same part** of the **same file** differently in the two branches you're merging, Git **won't be able** to merge them cleanly:

```
$ git merge testing
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
```

You can use the **status** command to see which files have conflicts:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified: README

no changes added to commit (use "git add" and/or "git commit -a")
```

Resolving Conflicts

Editing the file with conflicts we can see the conflict:

```
This is a README file
<<<<< HEAD
This was added in the master branch
=====
This was added in the testing branch
>>>>> testing
```

To solve it we just have to edit the file:

```
This is a README file
This was added in the master branch
This was added in the testing branch
```

And commit the merge:

```
$ git commit
```

Git Ignore

- A *.gitignore* file specifies intentionally untracked files that **Git should ignore**. Files already tracked by Git are not affected.
- Each line in a *.gitignore* file specifies a **pattern**.
- Some examples:

```
# this is a comment
docs/          # everything inside root directory docs
**/docs/       # any docs directory
!docs/**/*.* # don't ignore (!) any .txt files inside directory docs
```

What files to ignore: 1) not used by your project, 2) not used by anyone else and 3) generated by another process.

Remotes

Remotes

Remote repositories are **versions** of your project that are hosted **elsewhere** (another folder, the local network, the internet, ...).

You can **push** and **pull** data to and from remotes but first you need to learn how to configure them properly.

Cloning

The easiest way to end up with a remote, is to **clone** another repository.

```
$ git clone https://example.com/test-repository
Cloning into 'test-repository'...
remote: Enumerating objects: 129, done.
remote: Counting objects: 100% (129/129), done.
remote: Compressing objects: 100% (73/73), done.
remote: Total 129 (delta 54), reused 115 (delta 44), pack-reused 0
Receiving objects: 100% (129/129), 46.90 KiB | 565.00 KiB/s, done.
Resolving deltas: 100% (54/54), done.
```

To list our remotes (the verbose **-v** flag gives us some info about the URL):

```
$ git remote -v
origin  https://example.com/test-repository (fetch)
origin  https://example.com/test-repository (push)
```

We can see that git named our remote **origin** and set it up for both **fetching** and **pushing** data.

Protocols

Git can use **four** major network protocols to transfer data to and from **remotes**:

- **Local** - Useful if you have access to a shared mounted directory.
- **Git** - A special daemon that comes packaged with Git. SSH but without authentication or encryption.
- **SSH** - The most commonly used protocol.
- **HTTP** - Easiest to setup for read-only scenarios but very slow.

Adding Remotes

Besides the **origin** remote from where we **cloned** our project, we can **add** more remotes:

```
git remote add john http://john-laptop.org/test-repository
```

In this example, we added a new remote and gave it the alias *john*:

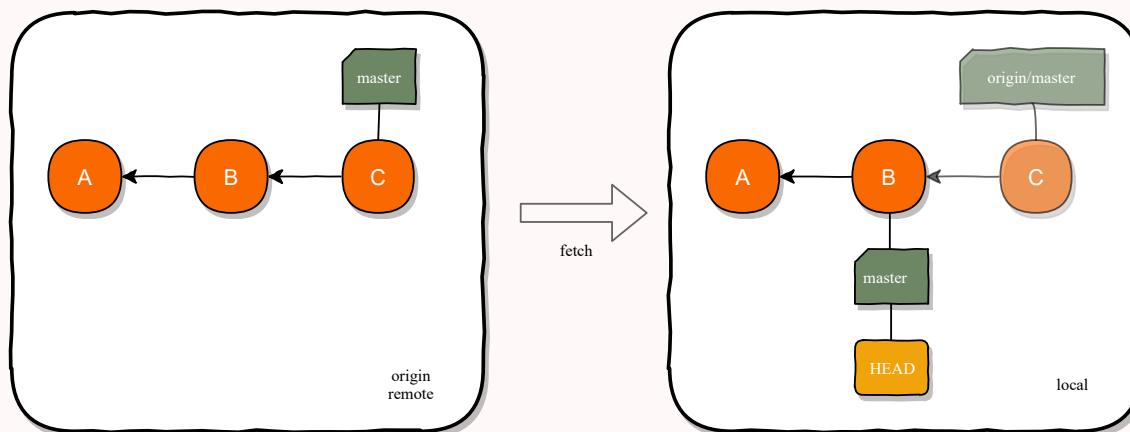
```
$ git remote -v
origin https://example.com/test-repository (fetch)
origin https://example.com/test-repository (push)
john http://john-laptop.org/test-repository (fetch)
john http://john-laptop.org/test-repository (push)
```

Fetching

Fetching **gets** all the data from a **remote** project that you don't have yet.

After fetching, you will also have **references** to all the **branches** from that remote.

```
$ git fetch origin
```



Fetching **only downloads** the data to your local repository. It doesn't automatically merge it with any of your work or modify what you're currently working on.

Tracking Branches

Tracking branches are local branches that have a **direct relationship** to a remote branch.

When you clone a repository, it generally **automatically creates a master branch that tracks origin/master**.

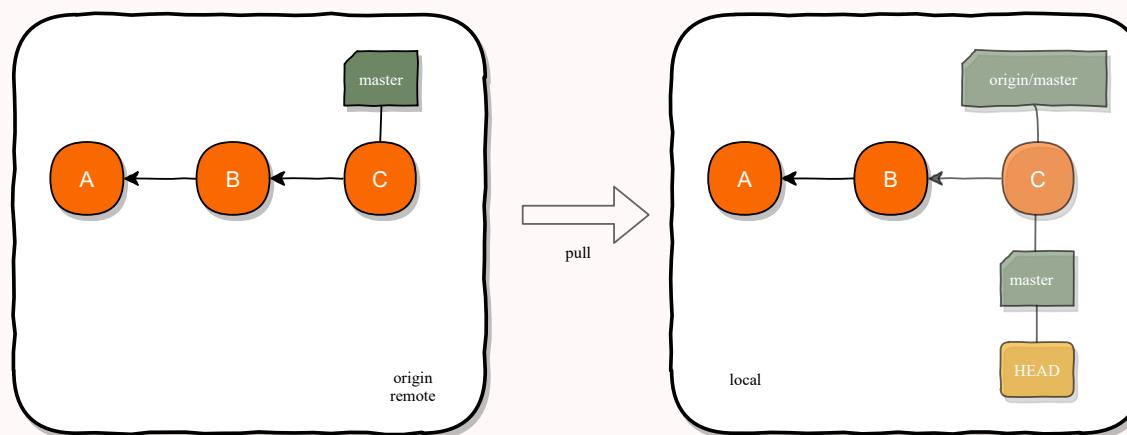
You can set up other tracking branches:

```
$ git checkout --track origin/feature # creates a local feature branch  
# that tracks origin/feature
```

Pulling

If your current branch is set up to **track** a remote branch, you can use the **git pull** command to automatically **fetch** and then **merge** that remote branch into your current branch.

```
$ git pull origin master # fetches and merges origin/master
```



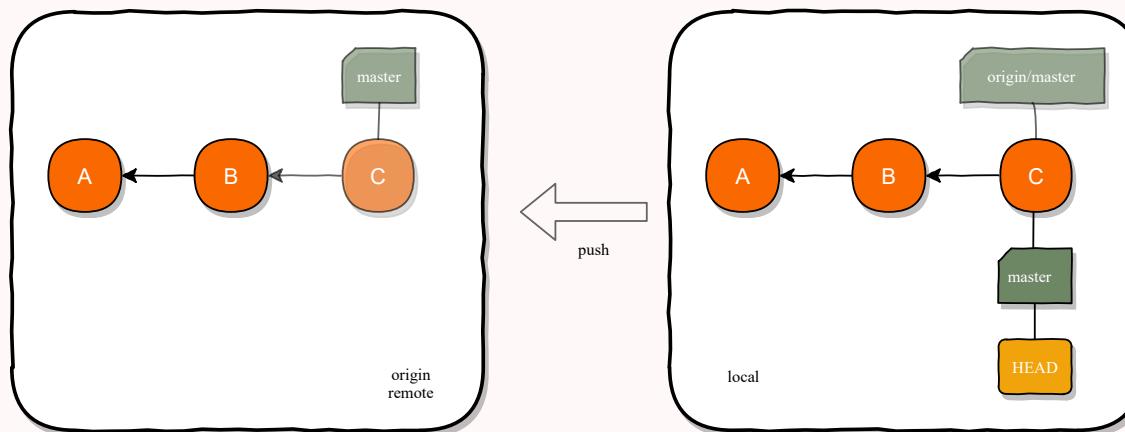
This fetches data from the server you originally cloned from and automatically tries to merge it into the code you are currently working on.

```
$ git pull # uses default values for current local branch
```

Pushing

Pushes local modifications to a remote. Only fast-forward merges are allowed so you might need to fetch and merge locally first.

```
$ git pull # or git pull origin master  
$ echo "some changes" >> README  
$ git commit -a -m "Made some changes"  
$ git push # or git push origin master
```



With the **-u** flag, it also sets the local branch to track the remote branch.

Git Hosts

Some free (for open source, education and small projects) git hosts you can use:

- GitHub
- BitBucket
- GitLab
- SourceForge

Reverting

Reset

The **reset** command resets the current branch HEAD to a certain commit.

These are some of the many different modes it can operate under:

- **--soft** - Does not touch the index¹ file or the working tree at all.
- **--hard** - Resets the index and working tree.
- **--mixed** - Resets the index but not the working tree (**default mode**).

¹ The staging area.

Local unstaged changes

If you **haven't staged or committed** the changes you want to revert you can:

```
$ git checkout -- README # undo changes to a single file
```

```
$ git reset --hard      # discard all local changes
```

Staged but uncommitted changes

If you have **staged the changes** you want to revert but **haven't committed them yet**, you can:

```
$ git reset HEAD <file> # unstage changes to a single file
```

```
$ git reset # unstage all changes
```

Committed but not pushed

If you have **already committed** the changes you want to revert but **haven't pushed** them to a remote yet, you can find the *commit-id* you want to revert to and:

```
$ git reset --hard <commit-id>
```

Committed and already pushed

You should try, **really hard**, to never rewrite public history.

For that reason, if you want to revert a file that was already pushed, your best bet is to use **revert**:

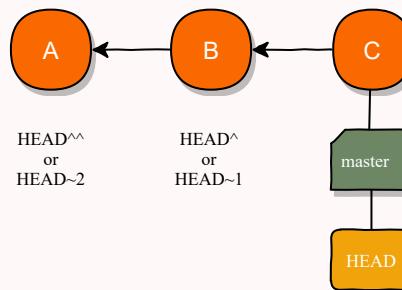
```
$ git revert <commit-id>
```

This will introduce the changes needed to revert the ones done by the commit without deleting the commit from history.

Relative commits

The ~(tilde) and ^(caret) symbols are used to point to a position **relative** to a specific commit.

- COMMIT[^] refers to the **previous** commit to COMMIT.
- COMMIT^{^^} refers to the **previous** commit to COMMIT[^].
- COMMIT~2 refers to **two commits** before to COMMIT.
- And so on...



Workflows

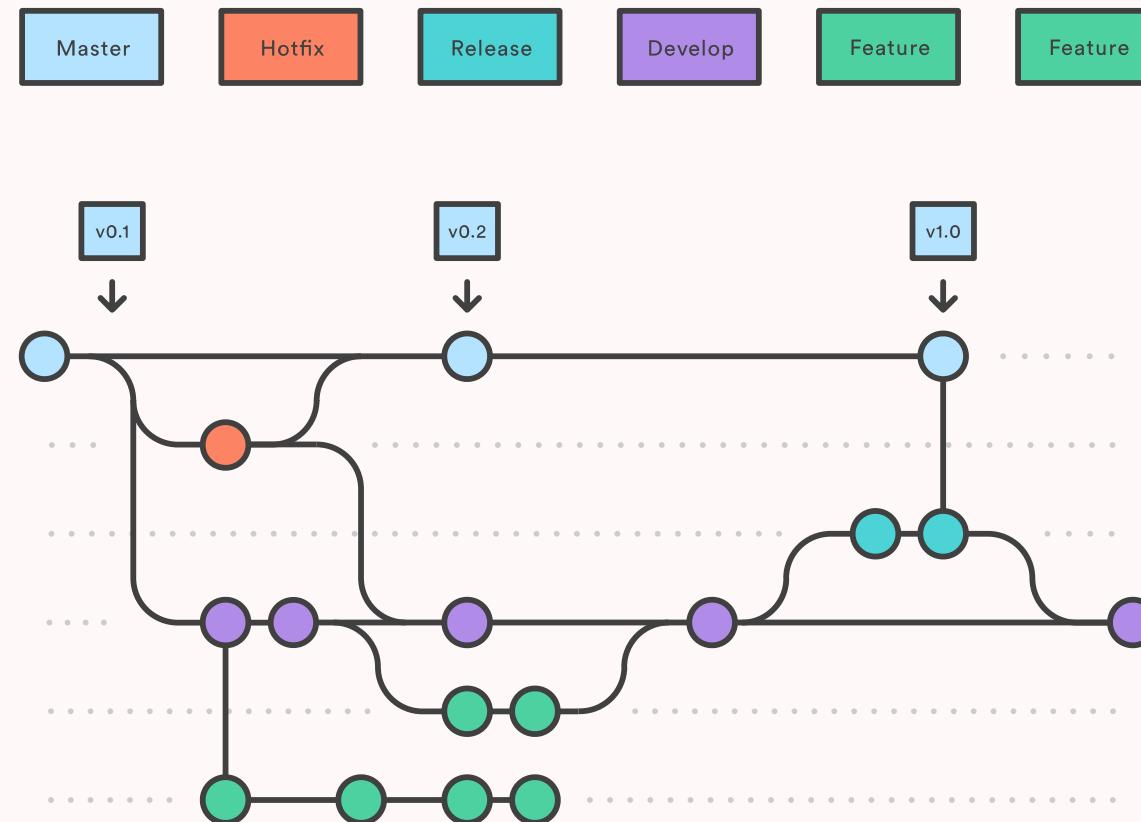
Workflows

There are endless different ways to use Git. For example:

- Having **feature branches** for each new feature.
- Having **release branches** where releases can be maintained.
- Hot fix branches to quickly patch production releases.

Git Flow

Git Flow is one way, but not the only one, of using git.



What's important is that you, and your team, are **consistent** in the way you use Git.

More

More stuff

Things we haven't talked about:

- **Tags** - Really just unmovable branches. Useful for marking releases.
- **Rebase** - A different way to merge.
- **Hooks** - IFTTT for Git.
- **Blame** - Who broke the code?
- **Bisect** - Finding a bad commit.
- **Stash** - Save these changes for later.
- **Pull requests** - Please take my code...
- And so much more...

Java

André Restivo

Index

Introduction

Basics

Arrays

OOP

Classes

Objects

Inheritance

Methods to Override

Garbage Collection

Packages

Exceptions

Collections

Threads

Input/Output

Introduction

What is Java?

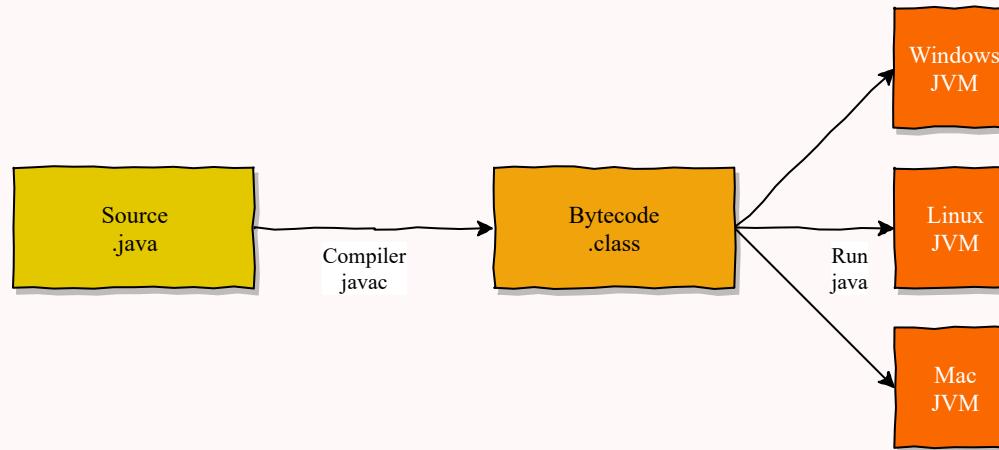
- Designed by: James **Gosling** (1995)
- Created by Sun Microsystems now owned by **Oracle**.
- Java is **open source** (under the GPL).
- Key **characteristics**:
 - General-purpose
 - Object-oriented (class-based)
 - Automatic memory management
 - Write-once / Run-everywhere

Java Editions

- **Java Card** - Smart cards and similar small memory footprint devices.
- **Java ME** - Micro Edition for **embedded** and **mobile** devices (IoT).
- **Java SE** - Standard Edition for regular Java applications. Mainly **desktop** and command-line apps.
- **Java EE** - Enterprise Edition for enterprise-oriented applications and servlets. Mainly large-scale **web**-oriented applications.

Compiling and Running

- Compiled into **bytecode** using the command **javac**.
- Bytecode can be run, using the command **java**, in any OS, as long as there is a Java Virtual Machine (**JVM**).
- Compiling and executing can be done using the Java Development Kit (**JDK**).
- The Java Runtime Environment (**JRE**) can be used instead for executing only.



Resources

- Book: Thinking in Java, 4th edition
- Book: Java Programming
- JDK 11 Documentation

Basics

Types

- Java is **strongly typed** so every variable must have a type.
- Java is **not a pure OOP language** so variables can have a **primitive** type or be a reference to an **object**.
- In Java, **arrays** are objects.
- There are no pointers but:
 - Primitive variables are stored as **values**.
 - Objects are stored as **references**.

Primitive Types

Primitive type are the most basic data types in Java.

| Types | Size (bits) | Minimum Value | Maximum Value | Precision |
|---------|-------------|---------------|------------------------------|---|
| byte | 8 | -128 | 127 | From +127 to -128 |
| char | 16 | 0 | $2^{16}-1$ | All Unicode characters |
| short | 16 | -2^{15} | $2^{15}-1$ | From +32,767 to -32,768 |
| int | 32 | -2^{31} | $2^{31}-1$ | From +2,147,483,647 to -2,147,483,648 |
| long | 64 | -2^{63} | $2^{63}-1$ | From +9,223,372,036,854,775,807 to -9,223,372,036,854,775,808 |
| float | 32 | 2^{-149} | $(2-2^{-23}) \cdot 2^{127}$ | From 3.402,823,5 E+38 to 1.4 E-45 |
| double | 64 | 2^{-1074} | $(2-2^{-52}) \cdot 2^{1023}$ | From 1.797,693,134,862,315,7 E+308 to 4.9 E-324 |
| boolean | — | — | — | false, true |
| void | — | — | — | — |

Literals

Java Literals are syntactic representations of boolean, character, numeric, or string data.

- **Boolean:** true or false.
- **Character:** 16-bit characters inside single quotes ('a'). Can be cast to int or long.
- **String:** Inside double quotes ("Java").
- **Integer:** Decimal (1234), Octal with a leading zero (02322), hexadecimal starting with ox (0x4D2) or binary starting with oB (0B10011010010). Ending with L if we want a long type integer (1234L).
- **Floating Point:** Ending with F or D for single and double precision (double is the default). Can be a decimal fraction or an exponential notation (0.1234 or 1234E-4).

Variables

Local variables are created by:

- giving it a **unique name**; and
- assigning it a **data type**.

```
int
```

Local variables must be given a value explicitly before being used:

```
int
```

This can be done in a single statement:

```
int
```

Conditional Blocks

Java has all the conditional blocks you would expect from a **C-family** programming language:

```
if  
  
else
```

And also:

```
switch  
case  
break  
case  
break  
default  
break
```

Loop Blocks

Loop blocks are also the expected ones. The **while-loop**:

```
while
```

Also a **do-while** variant:

```
while
```

And, of course, the **for-loop**:

```
for int
```

Operators

Arithmetic and boolean operators are also very similar to other C-family languages:

- Assignment:
- Numerical:
- Relational:
- Boolean:
- Bitwise:
- Tertiary:
- Type casting:

! Be careful with the `==` operator. It compares primitive types by value; but **compares objects by reference**.

Standard Input and Output

Writing to the screen can be accomplished using one of two methods:

```
new
```

```
int
```

Naming Convention

Names should follow the standard naming convention:

| Type | Form | Capitalization | Example |
|--------------------|------|--|---------------|
| Class or Interface | Noun | First word letter capitalized | PoliceCar |
| Methods | Verb | First word letter capitalized (except first one) | turnSirenOn() |
| Variables | — | First word letter capitalized (except first one) | carPlate |
| Constants | — | Uppercase with underscores separating words | MAX_SPEED |
| Packages | — | Starting with top-level domain, lowercase separated by periods | com.lpoo.util |

Strings

- In Java Strings are **immutable**, so they cannot be modified once created.
- String are a class defined in the *java.lang package* (more on that later):

The `+` operator **concatenates** strings:

```
String s1 = "Hello";  
String s2 = "World";  
String s3 = s1 + " " + s2;  
System.out.println(s3);
```

String are objects, so to compare them we **must** use the **equals** method:

```
if
```

Hello World

In Java, **everything** must belong to a **class**.

That means our customary **Hello World** example looks like this:

```
public class HelloWorld  
public static void main
```

Don't worry too much about the syntax for now.

Arrays

Arrays

In Java, an **array** is an **object**. This object has a given type for the contained primitive types or objects (int, char, String, ...).

An array can be declared in several ways:

```
int  
int
```

These arrays have been declared but haven't been instantiated yet. We can do it in a few different ways:

```
new int  
new int  
int
```

The default value depends on the data type. For objects it's **null**, for numeric types it's **0**, for booleans is **false** and for chars it's '**\u0000**' (whose decimal equivalent is 0).

Using Arrays

The size of an array can be obtained by using the **length** attribute:

```
for int
```

A simpler way of **looping** over an array is:

```
for int
```

Multidimensional Arrays

Arrays can have more than one **dimension**:

```
int
```

Sub-arrays can even have **different lengths**:

```
int
```

OOP

Abstractions

All programming languages provide **abstractions**:

- **Assembly** is an abstraction of **machine-code**.
- **Imperative** programming is an abstraction of **assembly**.

But they force us to think about the structure of the **machine** and not the structure of the **problem**.

OOP provides an abstraction where **elements** of the problem are **objects** in the solution space.

OOP allows you to describe the problem in terms of the **problem**, rather than in terms of the **computer** where the solution will run.

Objects

Alan Kay¹ on the five pillars of *Smalltalk*:

1. **Everything** is an object.
2. A program is a bunch of objects telling each other what to do by sending **messages**.
3. Each object has its own **memory** made up of **other objects**.
4. Every object has a **type**.
5. All objects of a particular type can receive the **same messages**.

"An object has state, behavior and identity" — Grady Booch², 1994.

"An object is characterized by a number of operations and a state which remembers the effect of these operations" — Ivar Jacobson², 1996.

1. Inventor of the Smalltalk language.

2. Two of the developers of UML (together with James Rumbaugh).

Object Oriented Pillars (A PIE)

Data Abstraction:

*Clear separation between the **public interface** of a data type, and its concrete implementation.*

Polymorphism:

A single symbol can represent a multitude of different types.

Inheritance:

*Objects can **inherit properties** and **behaviors** from other objects.*

Encapsulation (2 different concepts):

*A mechanism that: (1) allows **restricting access** to some of the **object's components** (2) facilitates the **bundling of data** with the **operations** on that data.*

Classes

Classes

- All **objects**, while being **unique**, are also part of a **class** of objects that have **characteristics** and **behaviors** in common.
 - Objects that are identical — **except** for their **state** — are grouped together into **classes of objects**.
 - Classes **extend** the programming language by adding new **data types**.
-

- Each class is defined by its **interface**.
- The **interface** determines the **requests** that you can make for a particular object.
- An object **provides services** and can use **other objects'** services to accomplish it.

Classes in Java

- In Java, public classes must be declared in a file with the **same name** but with a **.java** extension.
- This means that a Java file can have, at most, **one** public class.
- For example, inside a file called **Light.java** you could have:

```
public class Light
```

- As this is a **public class**, it can be accessed from anywhere.

Fields

- Objects store data inside **fields** (also called *member variables*).
- Each object keeps its **own** storage for its fields.
- Ordinary fields are **not shared** among objects.

```
public class Light
    private boolean      false
    private int
```

- Usually, fields should be made **private** so they can be accessed only from inside the object they belong to.
- Objects from other classes can access them using the class *public interface* (methods).

Methods

- Methods are how we **communicate** with objects.
- When we **invoke** or **call** a method we are asking the object to carry out a **task**.
- Each method has a **name**, **input parameters**, a **return type** and a **visibility**.

```
public class Light
public void turnOn
    this      true

public void turnOff
    this      false

public void setLevel int
    this
```

Visibility

For a class:

- **public**: can be referenced anywhere in the **application**.
- **protected**: can be referenced only in the **package**.
- **private**: only in **nested** classes, can be accessed only in the **outer** class.

For a variable:

- **public**: can be referenced anywhere in the **application**.
- **protected**: can be referenced only in **sub-classes** and in the same **package**.
- **package** (no modifier / default): can be referenced only in the same **package**.
- **private**: can be accessed only in the **class** it is defined in.

For a method:

- **public**: can be called anywhere in the **application**.
- **protected**: can be called only in **sub-classes** and in the same package.
- **package** (no modifier): can be called only in the same **package**.
- **private**: can be called only in the **class** it is defined in.

Keyword *this*

- **this** is a reference to the current object — the object whose method or constructor is currently running.
 - You can treat the reference just like any other **object reference**.
-
- If you are calling a method from **within** another method of the **same class**, you **do not need** to use *this*.
 - If you are referring to a field from **within** a method of the **same class**, you **do not need** to use *this*. But you should, for **readability** purposes (and sometimes to avoid **ambiguity**).

Constructor

- Constructors are special methods that are used to create **new objects**.
- Constructors have the **same name** as the class.
- Constructors **do not** have an explicit return type — they **implicitly** return the type they are constructing.
- Constructors can be **overloaded**.

```
public class Light
public Light
    this      false
    this

public Light int
    this      false
    this
```

Constructor Chaining

- Within a constructor, you can use the **this** keyword to invoke **another** constructor in the **same** class.
- This has to be the **first statement** of the constructor.

```
public class Light
public Light
    this
```

```
public Light int
    this      false
    this
```

Setters and Getters

- Object fields are usually kept **private** to improve **encapsulation**.
- It is common to provide public **setter** and **getter** methods to access and modify the value of a private field.

```
public class Light
    private boolean
    private int

    public void setLevel int
        this

    public int getLevel
        return
```

Static

- **Static fields** belong to the class instead of a specific object.
- **Static methods** can only access the static context of the class.

```
public class Light
    private static int

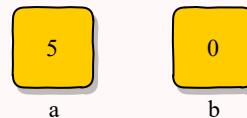
    public static int getMaximumLevel
        return
```

Objects

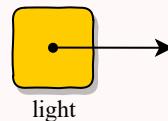
Objects

When a **primitive type** variable is declared, its value is **stored directly in its memory location**.

```
int
```



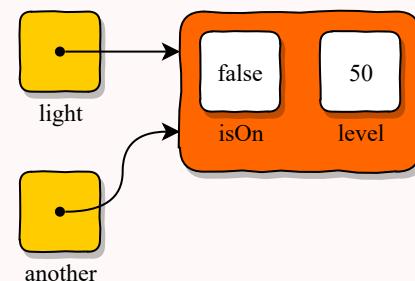
When an **object** is declared, it only contains a reference to the actual object.



Instantiation

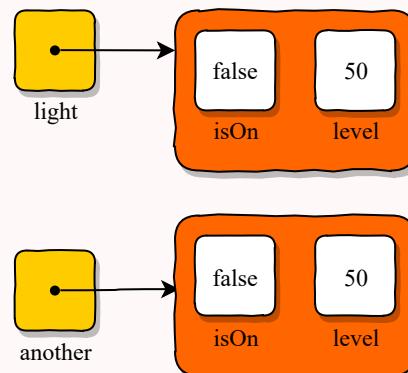
To create a new object, we just have to call its constructor using the **new** keyword:

```
new
```



Cloning

If we need to have two instances of the same object, we must use the `clone()` method. To use `clone`, our class must implement the **Cloneable** interface and override the `clone()` method making it **public**.



```
public class Light implements Cloneable  
  
    public      clone  throws  
    return super  
  
    new
```

Final

- The **final** keyword, allows us to declare **fields** and **variables** that **cannot be changed**.
- This only applies to the variable itself, so in the case of objects we can modify the **object** but not its **reference**.
- Can be used together with **static** to create **global constants**.

```
final           new  
  
    new
```

```
final int
```

```
public class Light  
private final static int
```

Parameters

Objects are passed to methods by **reference**; while **primitive** variables are passed by **value**.

```
private void change int
```

```
public void doSomething  
int  
new
```

Inheritance

Inheritance

- The mechanism of **basing** a class (or object) upon another class (or object), retaining a **similar** implementation.
- Inheritance should be used to establish a **is-a** relationship between classes.

-
- In Java, inheritance is **class-based**.
 - In Java, there is **no multiple-inheritance**.
 - In Java, if unspecified, all classes are based on the root **Object** class.

Extends

- The **extends** keyword allows a class to define a **different** superclass, inheriting all methods and fields from it.
- The **super** keyword allows calling a **constructor** from the superclass.
- You can **only** extend **one** class.

Tip: You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does.

Extends

```
public class Shape
    private

    public Shape
        this

public class Rectangle extends Shape
    private int

    public Rectangle int    int    int    int
        super

        this
        this
        this
        this
```

Overriding

Java allows classes to **override** superclass methods, providing that:

- The access modifier (visibility) for an overriding method can allow **more, but not less**, access than the overridden method.
- **Final** methods can not be overridden.
- **Static** methods can not be overridden.
- **Private** methods can not be overridden.
- The overriding method must have **same return type (or subtype)**.

We can call a parent class method using the **super** keyword.

```
public class Animal
    public void talk
    public final void eat

public class Dog extends Animal
    public void talk
        super
```

Abstract Classes

- Abstract classes cannot be instantiated but can be extended:
- They are used to:
 - Define methods which can be used by the inheriting subclass.
 - Define abstract methods which the inheriting subclass must implement.
 - Provide a common interface for their subclasses.

```
public abstract class Animal
    public abstract void talk

public class Dog extends Animal

    public void talk
```

Interfaces

- Java does not allow multiple-inheritance but it has **interfaces**.
- An interface is like a **fully abstract class** (only abstract methods).
- A class can implement **several interfaces**.
- Interfaces can be used in order to achieve **polymorphism**.

```
public interface Runner  public void run
public interface Walker  public void walk
public interface Eater   public void eat

public abstract class Animal implements Eater Walker
    public abstract void talk

public class Dog extends Animal implements Runner
    public void talk
    public void eat
    public void run
    public void walk
```

Polymorphism

In Java, a variable of a given type may be **assigned** a value of **any subtype**, and a method with a parameter of a given type may be **invoked** with an argument of **any subtype** of that type.

```
public void race  
  
public void main  
    new  
        new
```

Polymorphism

In Java, the method to be called is decided at **runtime**, based on the runtime type of the object.

```
public class Animal  
    public void talk
```

```
public class Dog extends Animal  
    public void talk
```

```
public class Cat extends Animal  
    public void talk
```

```
public void main  
    new  
    new  
    new
```

Methods to Override

Equals

As we have seen with **Strings**, when we want to compare objects we shouldn't use the `==` operator as it will only return true if the two objects are the same (have the same reference).

We should instead **override** the `equals(Object)` methods from the `Object` class.

The **correct** way to do so looks something like:

```
public boolean equals
    if this      return true
    if      null  return false
    if                  return false

    return
```

Hash Code

Another important method is the `hashCode()` method. This method should return the **same value** for two objects that are **equal**. So normally, when overriding the `equals(Object)` method you should also override the `hashCode()` method.

You can see the **hash code** at work in the `HashSet` data structure (which we will see in detail later on):

- When an element is **added**, the **hash code** is used to decide in which **bucket** it should be **stored**.
- When **searching** for an object, we only need to compare it (using `equals(Object)`) with objects in the same bucket.

Hash Code Implementation

To implement the `hashCode()` method, we should use a **subset** of the fields that are used in `equals(Object)`.

A possible implementation would be:

```
public int hashCode  
    return
```

To String

Another useful method from the **Object** class is the **toString()** method. This method returns a representation of any **Object** as a **String**. The default implementation is not very useful:

```
new
```

But we can **override** it and make it **better**:

```
public      toString
    return
```

So that we get:

```
new
```

Garbage Collection

Garbage Collection

- Automatic **garbage collection** is the process of looking at **heap** memory, identifying which objects are in use and which are not, and deleting the unused objects.
- An **in use** object, or a referenced object, means that some part of your program still maintains a pointer to that object.
- In Java, this process is done automatically so developers do not have to worry about **memory leaks**. **Or do they?**

Packages

Packages

- A **package** contains a **group** of classes, **organized** together under a single **namespace**.
- Classes in the **same package** can access each other's **package-private** and **protected** members.
- The package that a class belongs to is specified with the **package** keyword (first statement):

```
package
```

Packages are **stored** in the form of structured **directories**. For example: package "*com.example*" would be stored in directory "*com/example*".

Importing

To use a class from another package we must first import it:

```
import  
import  
  
    new
```

It is important to understand that **import** is simply used by the compiler to let you name your classes by their **unqualified** name.

Without the import statement this would still be valid:

```
    new
```

Exceptions

Exceptions

- When an error occurs within a method, the flow of execution of the program stops immediately, the method creates an **Exception** object and hands it off to the **runtime system**.
- The **runtime system** attempts to find something to **handle** it by following the **ordered list of methods** that have been called to get to the method where the error occurred.

```
public void someCode  
    null
```

```
public void moreCode
```

```
public void code  
try  
catch
```

Throw

The **throw** keyword is used to **explicitly** throw an exception (any sub-class of **Throwable**) from a method or any block of code. User defined exceptions typically extend **Exception** class.

```
public void someCode throws  
    throw new
```

```
public void moreCode throws
```

```
public void code  
try  
catch
```

Throws

If the **compiler** thinks there is a chance of rising an exception inside a method, then it will force us to either: 1) **catch** that exception, or 2) **declare** that we will **throw** that exception.

```
public void someCode() throws  
    throw new  
  
public void moreCode() throws
```

In this example, the **moreCode()** method is calling a method that **throws** an Exception, so it has to **throw** it also or **catch** it.

Finally

- The **finally** block always executes when the a try block exits.
- This ensures that the **finally** block is executed even if an **unexpected** exception occurs or an accidental return statement is added.
- Putting **cleanup code** in a **finally** block is always a **good practice**, even when no exceptions are anticipated.

```
public void code
try

catch

finally
```

Throw or Catch

The decision between **throwing** an exception and **catching** it might be an hard one:

- Methods should **catch** an exception if they can **handle** it locally.
- Methods should **throw** an exception if there is **nothing** they can do about it.

Catching an exception and **doing nothing** about it, besides printing the stack trace, is **always a bad idea**.

Collections

Collections

- A **Collection** is a **group** of individual objects represented as a **single unit**.
- Java provides the **Collection Framework** which defines several classes and interfaces to represent a group of objects as a single unit.
- The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main interfaces of Java Collection classes.

Collection Classes

- **Set** : Doesn't allow duplicates: HashSet (Hashing based), TreeSet (balanced BST based; implements SortedSet)...
- **List** : Can contain duplicates and elements are ordered: LinkedList (linked list based), ArrayList (dynamic array based), Stack, Vector, ...
- **Queue** : Typically order elements in FIFO order: LinkedList, PriorityQueue (not in FIFO order)...
- **Deque** : Elements can be inserted and removed at both ends: ArrayDeque, LinkedList...
- **Map** : Contains Key value pairs. Doesn't allow duplicates: HashMap and TreeMap (implements SortedMap).

Parameterized Collections

Java Collections are **parameterized** (using **Generics** — more about this later).

This means that we can define the **type of data** that the collection will **store**.

```
new  
new  
new
```

```
for
```

Notice that we used **List** instead of **ArrayList** to declare the variable. **List** is the **interface** that all lists **implement** and **ArrayList** is a **concrete instantiation** of that interface.

This is the "*Return the most specific type, accept the most generic type*" principle.

List

Some examples on how to use lists:

```
new  
new  
  
for
```

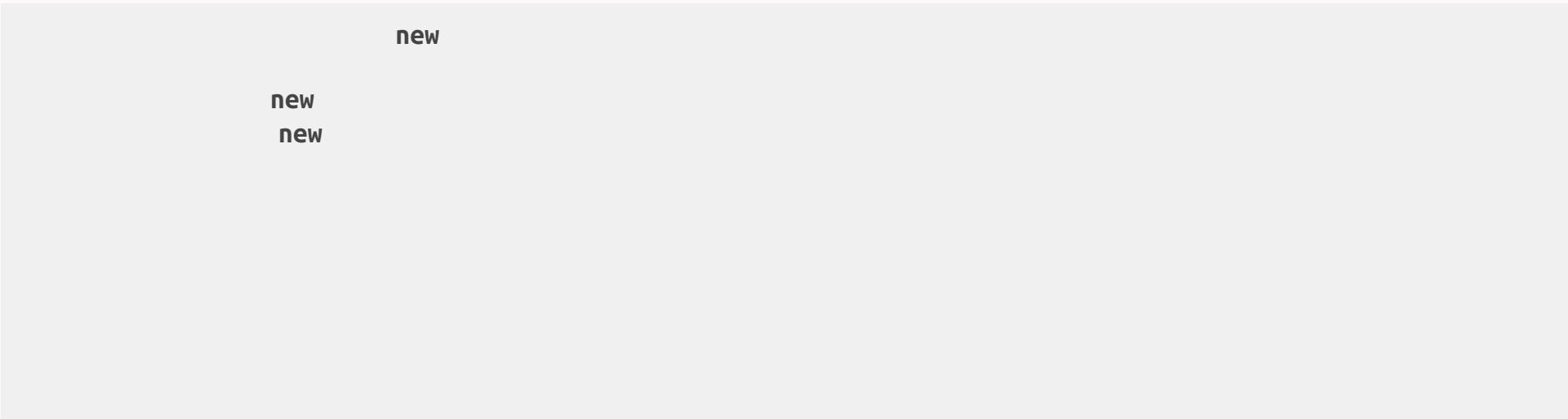
Set

Some examples on how to use **sets**:

```
new
```

Map

Some examples on how to use maps:



new

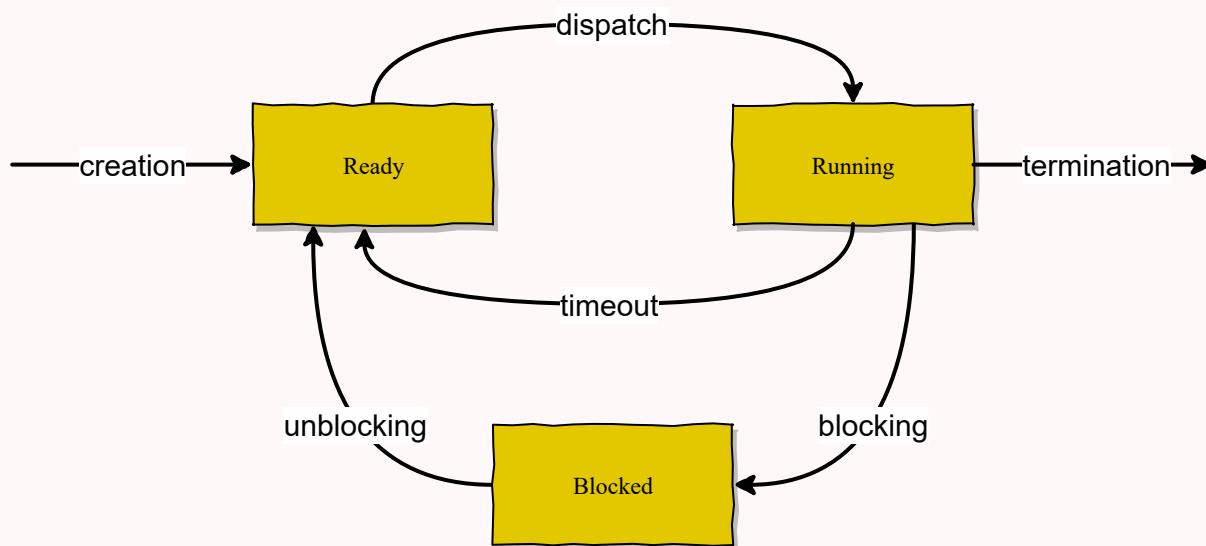
new

new

Threads

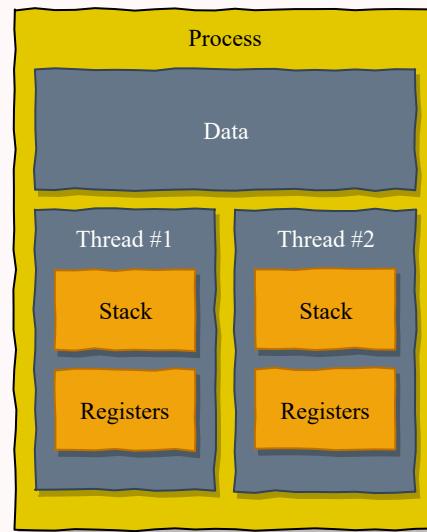
Processes

Multitasking is a method to allow **multiple processes** to share processors and other system resources.



Threads

A process may be made up of **multiple threads** of execution that execute instructions concurrently.



Threads are **lightweight processes** that have their own **stack** but have access to **shared data**.

Why threads?

Until now, all our code has been running in a **single main thread**.

But what happens if we need to block to read data from some source but still want our state and view to be updated?

```
public void run  
    while true
```

Threads in Java (1)

There are two different ways to create a new thread in Java.

- 1) Extend the **Thread** class and override the **run()** method:

```
public class GameUpdater extends Thread  
  
    public void run  
  
        new
```

Or just:

```
new  
  
    public void run
```

Threads in Java (2)

There are two different ways to create a new thread in Java.

2) Implement the **Runnable** interface and start a **Thread** with it:

```
class GameUpdater implements Runnable  
  
public void run  
  
new      new
```

Or just:

```
new      new  
  
public void run
```

Thread Class

The thread class has a series of useful methods:

- **void start()** - Causes this thread to **begin** execution; the Java Virtual Machine calls the **run** method of this thread.
- **static Thread currentThread()** - Returns a **reference** to the **currently executing** **thread** object.
- **long getId()** - Returns the **identifier** of this Thread.
- **void join()** - **Waits** for this thread to **die**.
- **boolean isAlive() and isInterrupted()** - Tests whether this thread is **alive** or has been **interrupted**.
- **static Thread interrupted()** - Checks if current thread has been interrupted and **resets flag**.

Interrupt

A thread cannot order another thread to stop. It has to ask nicely:

```
new  
  
public void run  
while true  
  
if           break
```

Sleep

The `Thread.sleep()` method can be used to pause the execution of current thread for specified time in milliseconds.

If the thread is interrupted during that time, an Exception is raised:

```
new  
  
public void run  
    while true  
  
    try  
  
        catch  
  
        break  
  
    if           break
```

Multi-Threading

Multi-threading programming can be tricky!

```
class Model
int
public void increment
```

```
class View
public void draw
```

new

new

```
new
```

```
public void run    while true
```

```
new
```

```
public void run    while true
```

Synchronized Blocks

- To make threads play nice with each other, we can use **synchronized** blocks.
- Synchronized blocks use a mechanism known as **monitor locks** (or intrinsic locks).
- A **synchronized block** uses an object as a lock.
- **No** two threads can enter a **synchronized block** if using the **same object** as a lock.

Synchronized Block Example

Each loop is synchronized on the **same object** (the Model **m**).

So, **v.draw()** will never be called while **m.increment()** is being executed.

```
new           new
```

```
new
public void run
while true
synchronized
```

```
new
public void run
while true
synchronized
```

Synchronized Methods

When a **synchronized method** is called, it automatically acquires the **intrinsic lock** for that method's object and releases it when the method **returns**.

```
class Model
    int
    public synchronized void increment
    public synchronized void draw
```

```
    new
```

```
new
public void run
while true
```

```
new
public void run
while true
```

Wait and Notify

Sometimes we need a thread to wait until something happens.

The `Object.wait()` method, pauses a thread until another thread calls `Object.notify()` on the same object.

Calls to wait and notify must be **synchronized**.

```
new  
  
public synchronized void run  
try  
  
catch  
  
synchronized
```

Input and Output

Streams

All fundamental I/O in Java is based on **streams**.

A stream represents a **flow of data** with a **writer** at one end and a **reader** at the other.

Abstract Streams

- **InputStream** and **OutputStream** are the basic **abstract** input and output stream for **unstructured bytes**.
 - All other byte streams are built on top of these two classes.
-
- **Reader** and **Writer** are the basic **abstract** input and output stream for **unicode chars**.
 - All other character stream are built on top of these two classes.

File Streams

FileInputStream, **FileOutputStream**, **FileReader** and **FileWriter** are implementations of **InputStream**, **OutputStream**, **Reader**, and **Writer** that read from and write to files on the local filesystem.

The **File** class represents a file in the local filesystem.

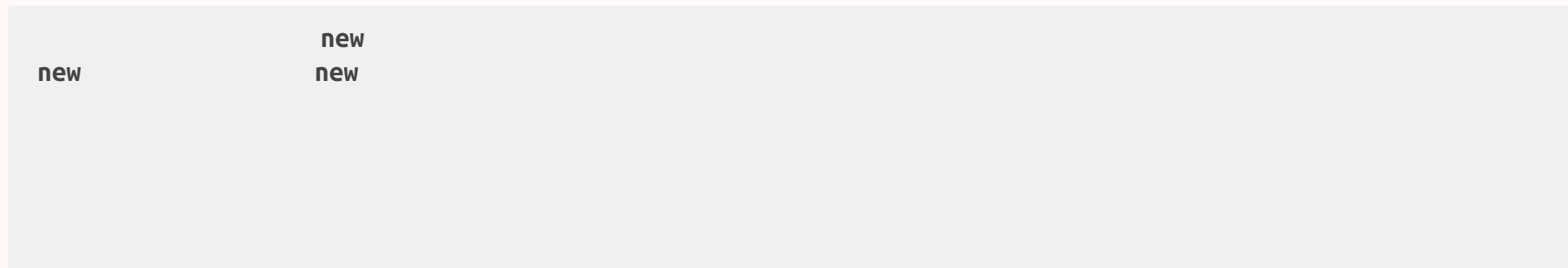
Examples:

| | | |
|------|------|-----|
| | new | new |
| int | | |
| | new | new |
| char | char | |

Bridge Streams

- **InputStreamReader** and **OutputStreamWriter** are classes that convert bytes to characters and vice versa.
- **DataInputStream** and **DataOutputStream** are specialized stream filters that add the ability to read and write primitive types.
- **ObjectInputStream** and **ObjectOutputStream** are stream filters that are capable of writing serialized Java objects and reconstructing them.

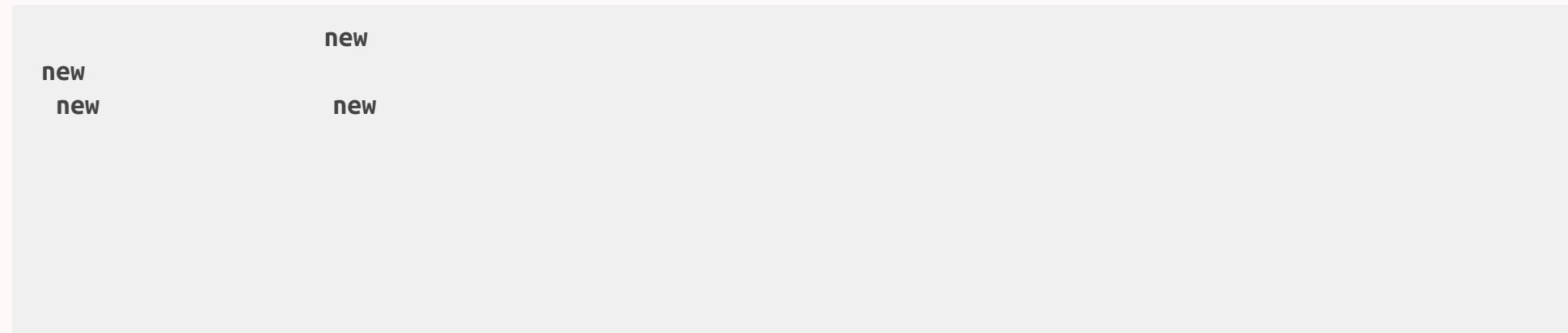
Example:



Buffered Streams

BufferedInputStream, **BufferedOutputStream**, **BufferedReader** and **BufferedWriter** add buffering capabilities to other streams. This increases efficiency.

Example:



Resources

Resources are pieces of data that **are part**, and can be **accessed**, from within a Java application.

When you create a **Java Gradle** project in **IntelliJ**, a folder for *resources* will be created inside both the **main** and **test src** folders.

To access them you can do something like this:

```
private static     readLines int      throws
                  new           new
                  new
for                   null
return
```

Unit Testing

For Java, Using JUnit, Mockito, and PIT

André Restivo

Index

Introduction

Unit Testing

JUnit

Test Isolation

Mockito

Code Coverage

Mutation Testing

Introduction

Software Testing

A process to evaluate the **quality** and **functionality** of a software system:

- Does the software meet the specified **requirements**, both **functional** and **non-functional**?
- Are there any **defects** (*aka* bugs)?

Software testing comes in **many forms** and can be done at **different levels** of the software development cycle.

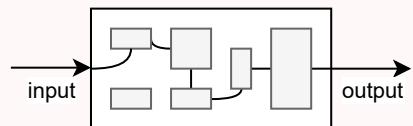
Automated Testing

Traditional software testing was done by **deploying** your application to a **test environment** and manually performing **black-box** tests. For example, by **clicking** through the **user interface** to find if something was **broken**.

Automated testing is a **technique** where the **tester/developer** writes **scripts** to test and compare the **actual** outcome with the **expected** outcome.

Black-box vs. White-box

In **black-box** testing, the actual **internal structure** of the item being tested is **unknown** or not taken into consideration.



In **white-box** testing, the design of the test cases is **based on the internal structure** of the system being tested, so that the **maximum number** of different **code paths** are covered.

Testing Levels

- **Unit Testing** - testing individual units of a software system in order to validate if they perform as designed.
- **Integration Testing** - individual units are combined and tested as a group in order to expose faults in the interaction between them.
- **System Testing** - the complete software system is deployed and tested to evaluate its compliance with the specified requirements.
- **Acceptance Testing** - the complete system is tested for acceptability to evaluate if it is compliant with the business requirements and acceptable for delivery.

Testing Types

- **Smoke** - ensure that the **most important** features work.
- **Functional** - verify if **functional requirements** are met.
- **Usability** - verify if the system is easily **usable** by end-users.
- **Security** - uncover **vulnerabilities** of the system.
- **Performance** - test the **responsiveness** and **stability** of the system under a certain load.
- **Regression** - ensure that previously developed and tested software still performs after a change.
- **Compliance** - determine the **compliance** of a system with any **standards**.

Unit Testing

Unit Testing

Testing individual units of a software system in order to validate if they perform as designed.

There are several **advantages** to unit tests:

- Increases confidence in changing/maintaining code.
- In order to make unit testing possible, codes need to be **modular**, which makes them more **reusable**. Good unit testing promotes good code.
- Development becomes **faster** as system, as a whole, does not need to be run to test newly written code.
- When a test fails we know **which unit** is the culprit.

FIRST

The **FIRST** principles of unit testing:

- **Fast** - Unit tests should be **fast** so we can run them often.
- **Isolated / Independent** - Only test **one unit** at a time. Only test **one thing** at a time. **Order** of tests should **not matter**.
- **Repeatable** - Results should be deterministic and not depend on the environment (time, available data, random values, ...).
- **Self-validating** - No manual checking necessary.
- **Thorough / Timely** - Cover every **use case** scenario (different from 100% code coverage). Test for **corner cases**, **large data sets**, **different roles**, **illegal arguments** and **bad inputs**...

The 3 As

A unit test should be divided into **three** different parts:

- Arrange - Where the test is **setup** and the data is **arranged**.
- Act - Where the the actual method under test is **invoked**.
- Assert - Where a **single logical assert** is used to test the outcome.

Helper classes can be used to **setup** data to be **reused** in **several** tests cases.

Test Doubles

Test doubles are pretend objects that help reduce complexity and verify code independently from the rest of the system. They come in many **flavours**:

- **Dummy** - never actually used; just to fill parameter lists.
- **Fake** - working implementations, but not suitable for production.
- **Stubs** - provide canned answers to calls made during the test.
- **Spies** - stubs that also record some information based on how they were called.
- **Mocks** - pre-programmed with expectations which form a specification of the calls they are expected to receive.

State vs. Behavior Testing

- **State Testing:** determine whether the exercised method worked correctly by examining the state after the method was exercised.
- **Behavior Testing:** specify which methods are to be invoked, thus verifying not that the ending state is correct, but that the sequence of steps performed was correct.

Spies and Mocks are usually needed for behavior testing.

JUnit

JUnit

JUnit is a testing framework for Java specialized in unit tests.

A JUnit test is a **method**, contained in a **class**, which is **only used for testing**.

A JUnit test must have the **@Test** annotation.

A simple **test class** looks like this:

```
import  
  
public class TestDog  
  
    public void testDogName  
        new
```

Asserts

JUnit provides a series of **assert methods** to help test for certain **conditions**:

- `fail([message])` - Fails the test.
- `assertTrue([message,] condition)`
- `assertFalse([message,] condition)`
- `assertEquals([message,] expected, actual)`
- `assertEquals([message,] expected, actual, tolerance)`
- `assertNull([message,] object)`
- `assertNotNull([message,] object)`
- `assertSame([message,] expected, actual)`
- `assertNotSame([message,] expected, actual)`

Message is an **optional message** specifying why the test failed.

Set Up and Tear Down

The `@Before` and `@After` annotations allows us to define methods that run **before** or **after** each test method.

These can be used to **setup** and **dispose** of any **data/classes** that are used by all tests, thus simplifying the **Arrange** phase.

There are also `@BeforeClass` and `@AfterClass` annotations that define methods that should be run only **once** for the **entire class**. These might help when test methods share a computationally **expensive** setup.

```
import  
  
public class TestDog  
    private  
  
    public void connectToDatabase  
        new  
  
    public void testDogRetrieval
```

Test Isolation

Test Isolation

One of the key features of **unit testing**, is that of test isolation. The whole point of **unit tests** is to **reduce the scope** of the system under test to a **small subset** that can be tested in isolation.

Most of the times this can be difficult without **changing our design**. For example, consider the following **class** and **test**:

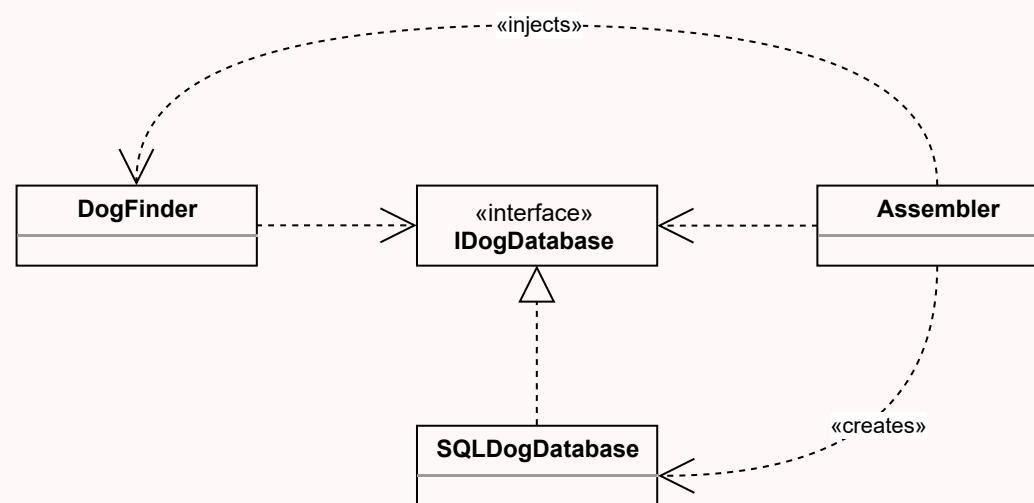
```
public class DogFinder
private          new
public         findBreed
               new
for
if
return
```

```
import
public class TestDogFinder
public void testDogRetrieval
new
for
if
```

Any test on the **DogFinder** class will depend on the **DogDatabase** class.

Dependency Injection

One way to achieve **test isolation**, is to use **Dependency Injection**. With this technique, **classes** no longer depend on other classes but **on interfaces**. The concrete instantiation of each interface is **injected** into the class by a third-party class (the **Assembler**).



Show me the Code

```
public interface IDogDatabase  
    public      getAllDogs throws
```

```
public class SQLDogDatabase implements IDogDatabase  
    public      getAllDogs throws
```

```
public class DogFinder  
    private  
  
    public DogFinder  
        this  
  
    public      findBreed      throws
```

```
public class Application  
    public static void main  
        try  
            new      new  
  
        catch
```

And now the Test

```
public class DogFinderTest
    class StubDatabase implements IDogDatabase

    public        getAllDogs  throws
                    new
                    new
                    new
    return

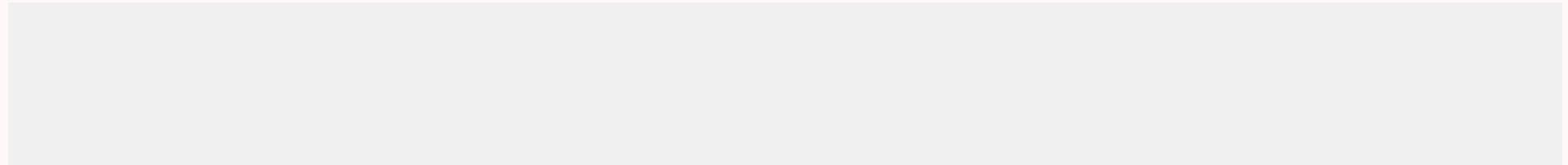
public void findBreed  throws
                    new
                    new
```

Mockito

Mockito

A simpler way to create **Mocks** and **Stubs** is to use a specialized framework like **Mockito**.

If we are using **Gradle**, the only thing we have to do to be able to use **Mockito** is add the dependency in our "build.gradle" file:



Mockito Stubs

Creating stubs with Mockito is very simple:

```
import  
  
public class DogFinderTest  
private  
  
public void setUp throws  
    new  
    new  
    new  
  
public void findBreed throws  
    new
```

When and Then

The `when` and `then*` keywords allows to configure Mockito stubs to return **canned answers** very **easily**:

```
    true  
    null
```

When the method returns void, the syntax is slightly different:

[When/Then Cookbook](#)

Verify

Until now we have been doing **state testing**. If we want to do **behavior testing** we need to use **mocks**, and **Mockito**, as the name implies, can **help us** with that.

```
public void findBreedCallsDatabaseOnlyOnce throws  
    new
```

[Verify CookBook](#)

Code Coverage

Code Coverage

- Measures the **number of code lines covered** by the **test cases**.
- Reports the **total** number of lines in the code and **number of lines executed** by tests.
- The **degree** to which the source code of a program is exercised when a **test suite** runs.
- The **higher** the code **coverage**, the **lower** the chance of having undetected software **bugs**.

But, code coverage doesn't tell the **whole story...**

Code Coverage Problems

- High coverage numbers are **too easy** to reach (we don't even need **asserts**).
- Good testing practices should result in **high coverage**. The inverse is not true.

So **why** do code **coverage** analysis:

- It helps us **find untested** parts of our source code that should be tested but are not.

Code Coverage in IntelliJ

In IntelliJ you can **run your tests with coverage** to get a percentage of code covered per **class** and/or **package**, for **all test suites** or just for **a few**.

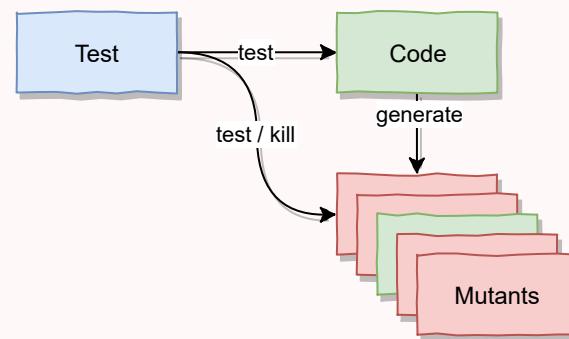
You also get **indicators** throughout your code showing which lines are tested and which are not.

Mutation Testing

Mutation Testing

A type of **software testing** where we **mutate** (change) certain statements in the **source code** and check if the test cases are able to **find** the errors.

The **goal** is to assess the **quality** of the **test cases** which should be **robust** enough to **fail** mutant code.

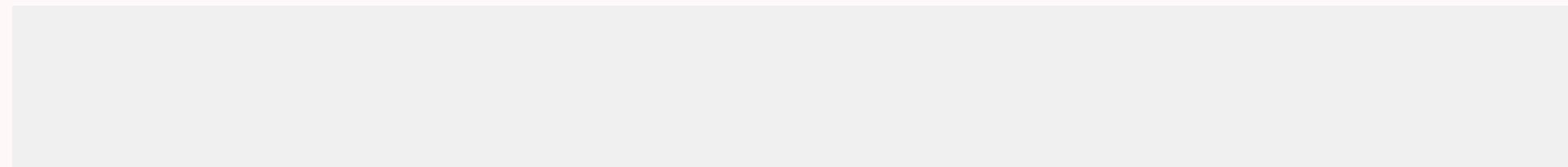


In the mutation testing lingo, **tests** are trying to **kill** as many **mutants** as possible (optimally 100% of them).

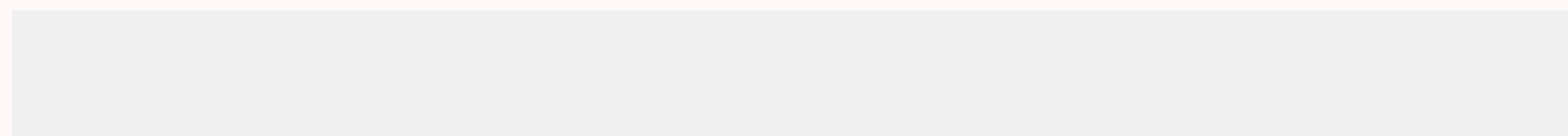
PIT Mutation Testing

PIT is a mutation testing system, providing gold standard test coverage for Java.

With **Gradle**, installing PIT for your project in **IntelliJ** is as easy as adding this second line to your **plugins** section in your "**build.gradle**":



PIT can be configured directly in your "**build.gradle**" using the same **command line parameters** as the command line version uses:



Target Classes

By default, PIT uses the group defined in the "`build.gradle`" file to automatically infer the `targetClasses` parameter. For example, if your "`build.gradle`" file has:

Then it will automatically infer the following:

Running Mutation Tests

PIT will automatically generate a **Gradle task** called "pitest". So you can **run mutations** tests simply by doing:

Reports will be created under "`build/reports/pitest/<timestamp>/`" in HTML format by default.

SOLID

...and other OO principles!

André Restivo

Index

Software Rot

Symptoms

Causes

SOLID Principles

Other Principles

Reference

Directly from Uncle Bob:

Martin, R.C., 2000. "Design principles and design patterns". *Object Mentor*, 1(34), p.597.

Software Rot

Software Rot

Even when software design **starts** as a pristine work of art, portraying the **clean** and **elegant** image in the **mind** of the designer, it **eventually** starts to **rot**:

- It starts with a **small hack** but the overall beauty of the design is still there.
- The hacks start **accumulating**, each one another **nail in the coffin**.
- The code **eventually** becomes an **incredibly hard to maintain mess**.

System Redesign

- At this point a **redesign** is needed. But the old code is still in **production, evolving and changing**.
- So the *system redesign* is trying to **shoot at a moving target**.
- **Problems** start to **accumulate** in the new design **before** it is even **released**.

Symptoms of Rotting Design

Rigidity

- The **tendency** for software to be **difficult to change**.
- Every **change** causes a **cascade** of subsequent changes.

When software behaves this way, managers **fear** to allow engineers to **fix** non-critical problems (as they may disappear for long periods of time).

Fragility

- The **tendency** of software to **break in many places** every time it is changed.
- Often in areas that have **no conceptual relationship** with the area that was changed.

When software behaves this way, managers and customers start to suspect that the **developers have lost control** of their software.

Immobility

- The **inability to reuse software** from other projects or from parts of the same project.
- The **work and risk required to separate** the desirable parts of the software from the undesirable parts are **too great** to tolerate.

Software ends up being **rewritten**.

Viscosity

Viscosity of the **design**:

- There is **more than one** way to make a change: preserving the **design**, and **hacks**.
- The **design** preserving methods are **harder** to employ than the **hacks**.

Viscosity of the **environment**:

- The development environment is **slow** and **inefficient** (long compile times, complicated and long check in procedures, ...).
- Developers end up choosing solutions that require **as few changes** as possible, **regardless** of whether the **design** is **preserved**.

Causes of Rotting Design

Changing Requirements

- Requirements change in ways that the **initial design did not anticipate**.
- Often **changes are urgent**, and **hacks** are used to make them; even if it **deviates from the original design**.

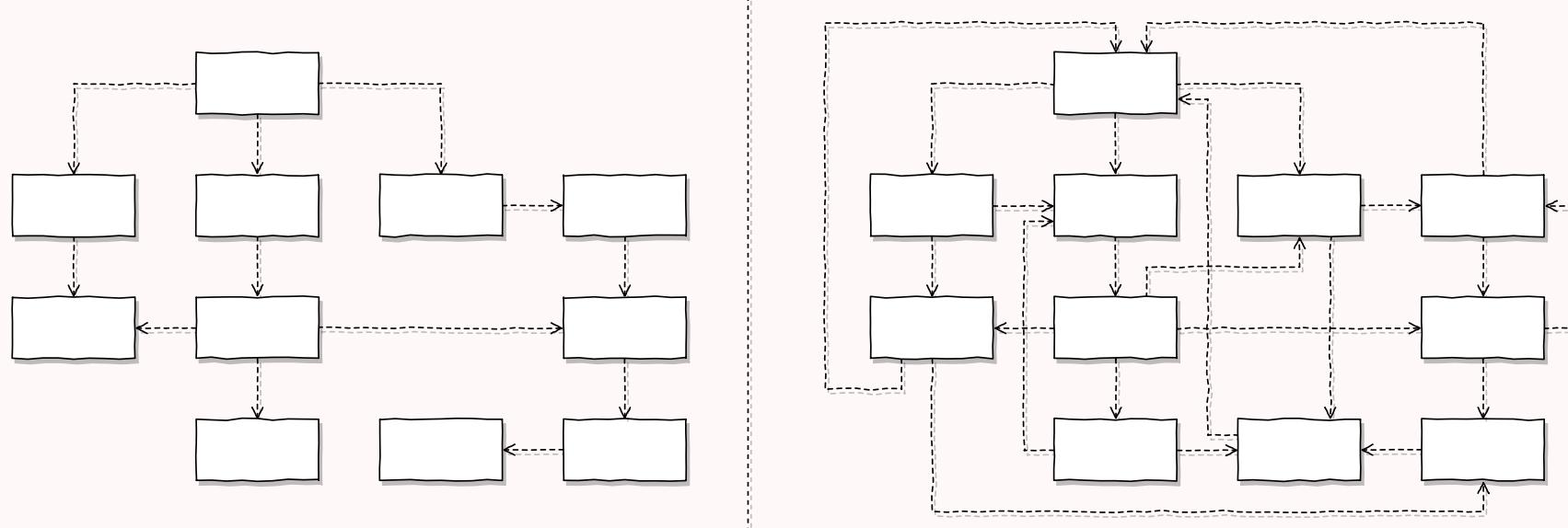
Changing requirements should **not be a surprise**, and blaming them is the **easy way out**:

- The **system design** must be **resilient** to these changes from the start.

Dependency Hell

If we analyze the four symptoms of rotting design just presented, carefully, there is one common theme among them: **improper dependencies** between modules.

- The **initial design** properly separates the **responsibilities** of each module; dependencies seem **logic and stratified**.
- As **time goes by**, **hacks** (needed because of **unforeseen requirement changes**), introduce **unwanted dependencies**.



Principles of Object-Oriented Design

SOLID

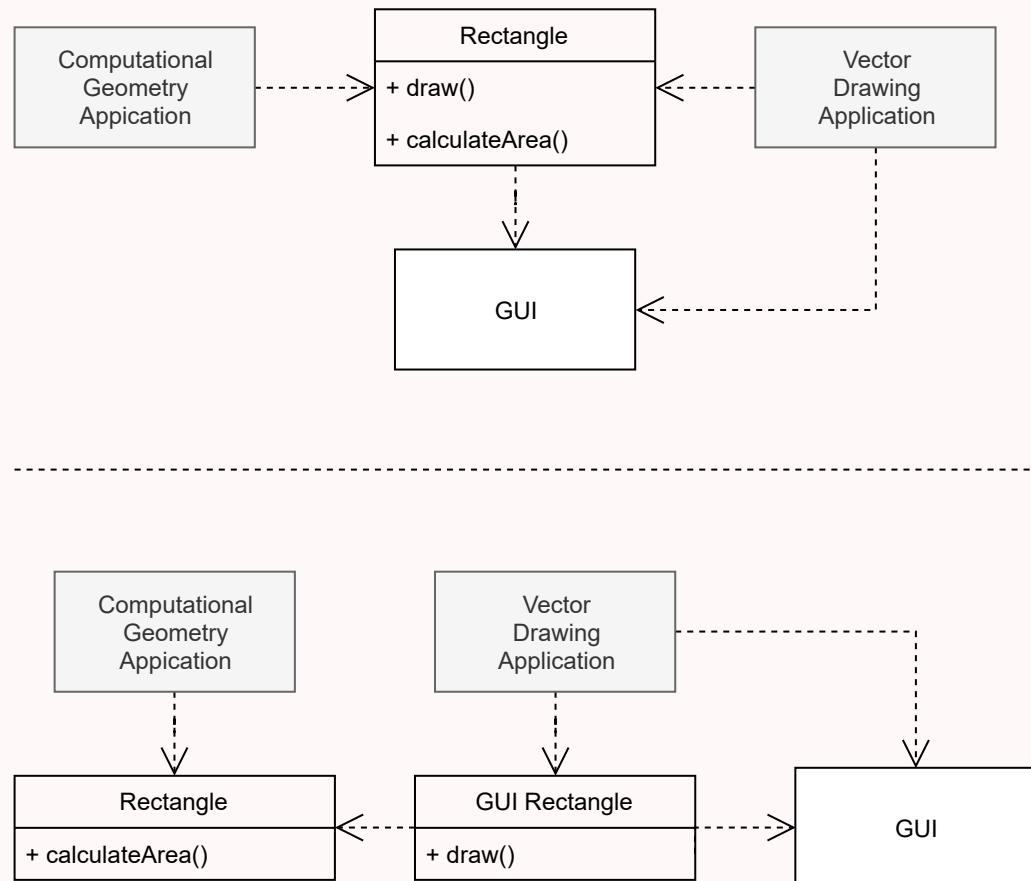
(S) The Single Responsibility Principle (SRP)

 "Each software module should have **one and only one reason** to change."

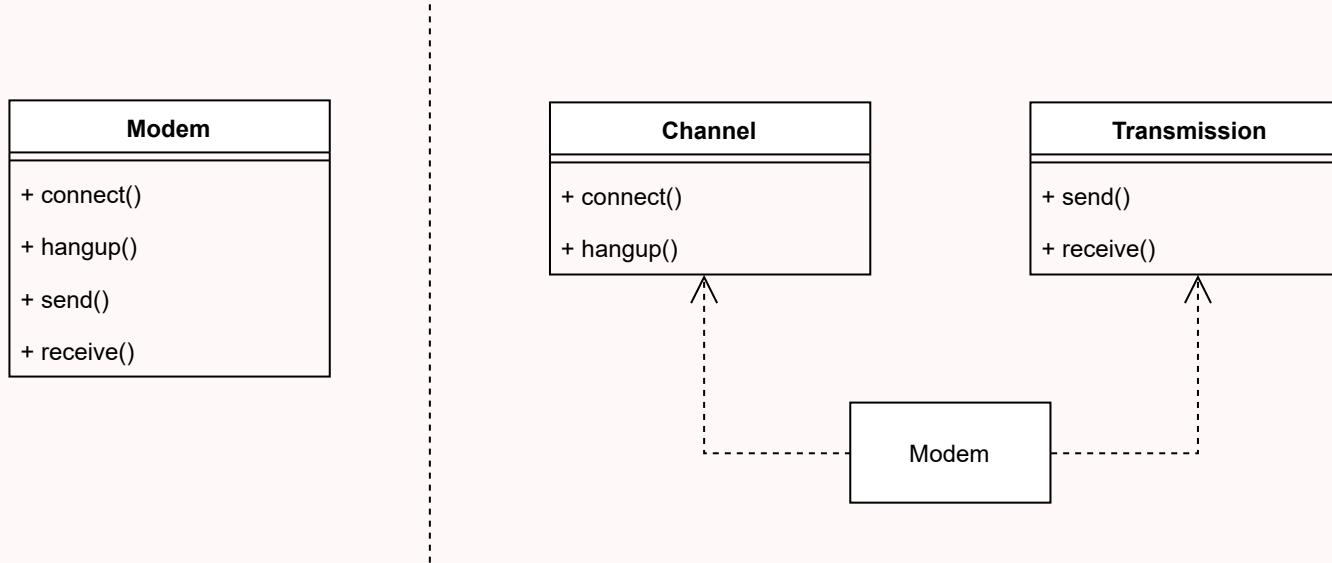
If a module assumes more than one responsibility, then:

- There will be **more than one reason** for it to **change**.
- **Changes** to one responsibility may **impair** the **ability** to meet the **others**.
- It might **force unwanted** and **unneeded dependencies**.

Example



When to (or not to) use?



- If the application is **not changing** in ways that cause the two responsibilities to **change at different times**, then there is **no need to separate them**.
- It is **not wise** to apply the SRP if there is **no symptom** (needless complexity).

Hiding Difficult Decisions

Parnas, D.L., 1972. *On the criteria to be used in decomposing systems into modules*. Communications of the ACM, 15(12), pp.1053-1058.

“We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of **difficult design decisions** or design decisions which are **likely to change**. Each module is then designed to **hide such a decision from the others**.”

(0) The Open-Closed Principle (OCP)

A module should be **open** for extension but **closed** for modification.

- We should **write** our modules so that they **can be extended**, without requiring them to be **modified**.
- So we can **add** new features to **existing** code, by only **adding** (and not modifying) new code.

Example

There can be many different types of shapes:

```
public class Shape
    enum
    private

    public void draw
        switch
            case
            case
                break
                break
```

What happens when we want to add another shape?

Solution: Dynamic Polymorphism

```
public abstract class Shape  
    public void draw
```

```
public class Square extends Shape  
    public void draw
```

```
public class Circle extends Shape  
    public void draw
```

Other Solution: Static Polymorphism

Also known as **generics** (more on that later):

No need to **rewrite** the *List* class to use it with a different type.

(L) The Liskov Substitution Principle (LSP)

Subclasses should be substitutable for their base classes.

A **user** of a **base class** should **continue to function properly** if a **derivative** of that base class is **passed** to it.

This might seem **obvious** at first, but many times its **hard to detect** that this principle is being **broken**.

The Rectangle-Square Dilemma

All squares are rectangles with equal height and width.

```
public class Rectangle
    public void setWidth double
    public void setHeight double
    public double getArea

public class Square extends Rectangle
    public void setWidth double
        this          this

    public void setHeight double
        this          this
```

LSP Violation

A client should rightfully expect the following to hold:

```
public void doSomething
```

If this method really needs this to hold, then it has to test if the Rectangle is really a Rectangle:

```
public void doSomething  
if instanceof
```

And we are back at the OCP problem!

LSP as Contracts

A derived class is **substitutable** for its **base** class if:

1. Its **preconditions** are no **stronger** than the **base** class method.
2. Its **postconditions** are no **weaker** than the **base** class method.

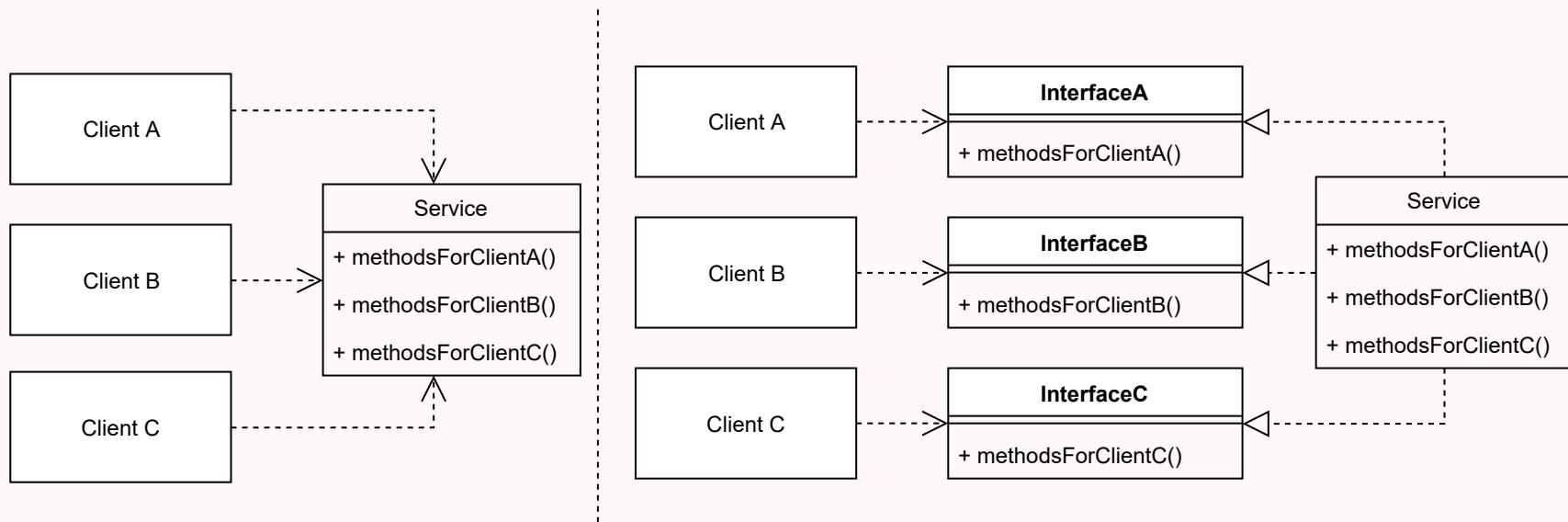
Or, in other words, **derived methods should expect no more and provide no less.**

(I) The Interface Segregation Principle (ISP)

Many client specific interfaces are better than one general purpose interface.

- Clients should **not** be forced to **depend** upon interfaces that they **do not use**.
- Clients should be **categorized** by their **type**, and **interfaces** for each **type** of client should be **created**.
- If **two or more** different client types **need the same method**, the method should be **added** to **both** of their interfaces.

One Service, Different Interfaces



- Makes the code more **readable** and **manageable**.
- Promotes the **single responsibility principle (SRP)**.

(D) The Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions.

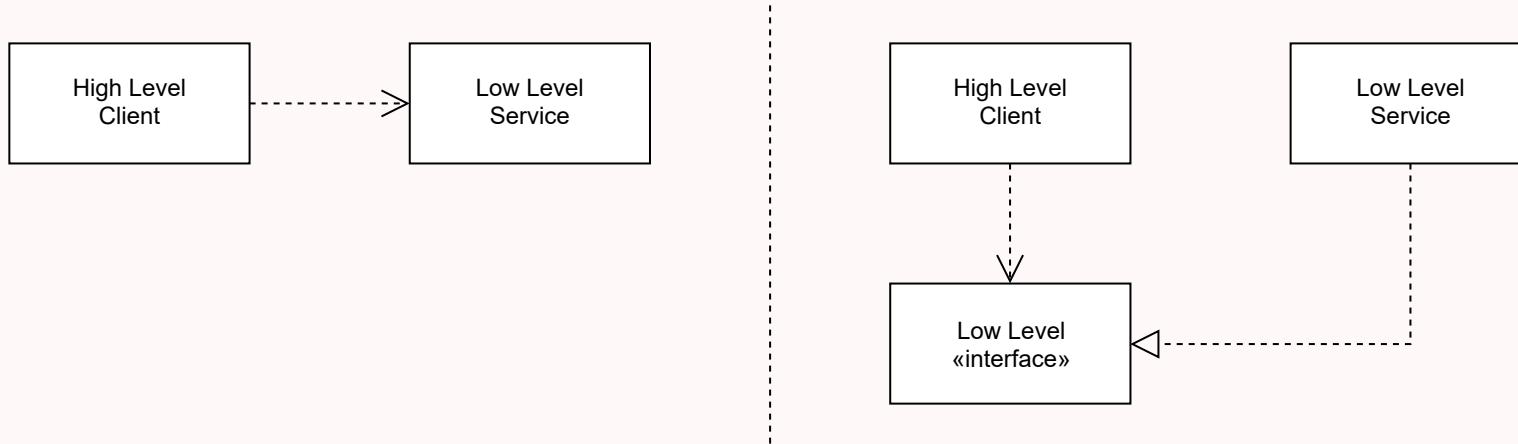
And

Abstractions should not depend on details. Details should depend on abstractions.

- We are **not just** changing the direction of the dependency.
- We are **splitting** the dependency by putting an **abstraction** in the **middle**.

Why?

Concrete things change a lot, abstract things change much less frequently.



- No client code has to be **changed** simply because an object it **depends on** needs to be **changed to a different one** (loose coupling).
- Promotes **testability**.
- Promotes **replaceability**.

Other Principles

Principles of Package Architecture

The Release Reuse Equivalency Principle (REP)

The granule of reuse is the granule of release.

- Code should **not be reused** by **copying** it from one class and **pasting** it into another.
- Only **components** that are **released** through a **tracking system** can be **effectively reused**.

The Common Closure Principle (CCP)

Classes that change together, belong together.

- If the code in an application **must change**, changes should be focused into a **single package**.
- If two classes almost always **change together**, then they **belong in the same package**.

Maintainability!

The Common Reuse Principle (CRP)

Classes that aren't reused together should not be grouped together.

- Generally **reusable** classes **collaborate** with other classes that are part of the **Reusable abstraction**.
- These classes **belong** in the **same package**.

Reusability!

The Package Coupling Principles

The Acyclic Dependencies Principle (ADP)

The dependencies between packages must not form cycles.

- The dependency graph should be a DAG (directed acyclic graph).
- Cycles in the dependency graph are effectively large packages.
- Cycles can be broken using the dependency inversion principle (DIP).

The Stable Dependencies Principle (SDP)

Depend in the direction of stability.

- **Stable** means "**hard to change**" (many clients), while **unstable** means "**easy to change**".
- Modules that are "**hard to change**" should **not depend** on modules that are "**easy to change**".
- The reason is that it makes the "**easy to change**" module "**harder to change**" because of the **impact** on the depending module.
- You need "**easy to change**" packages, or your software **cannot change easily**.

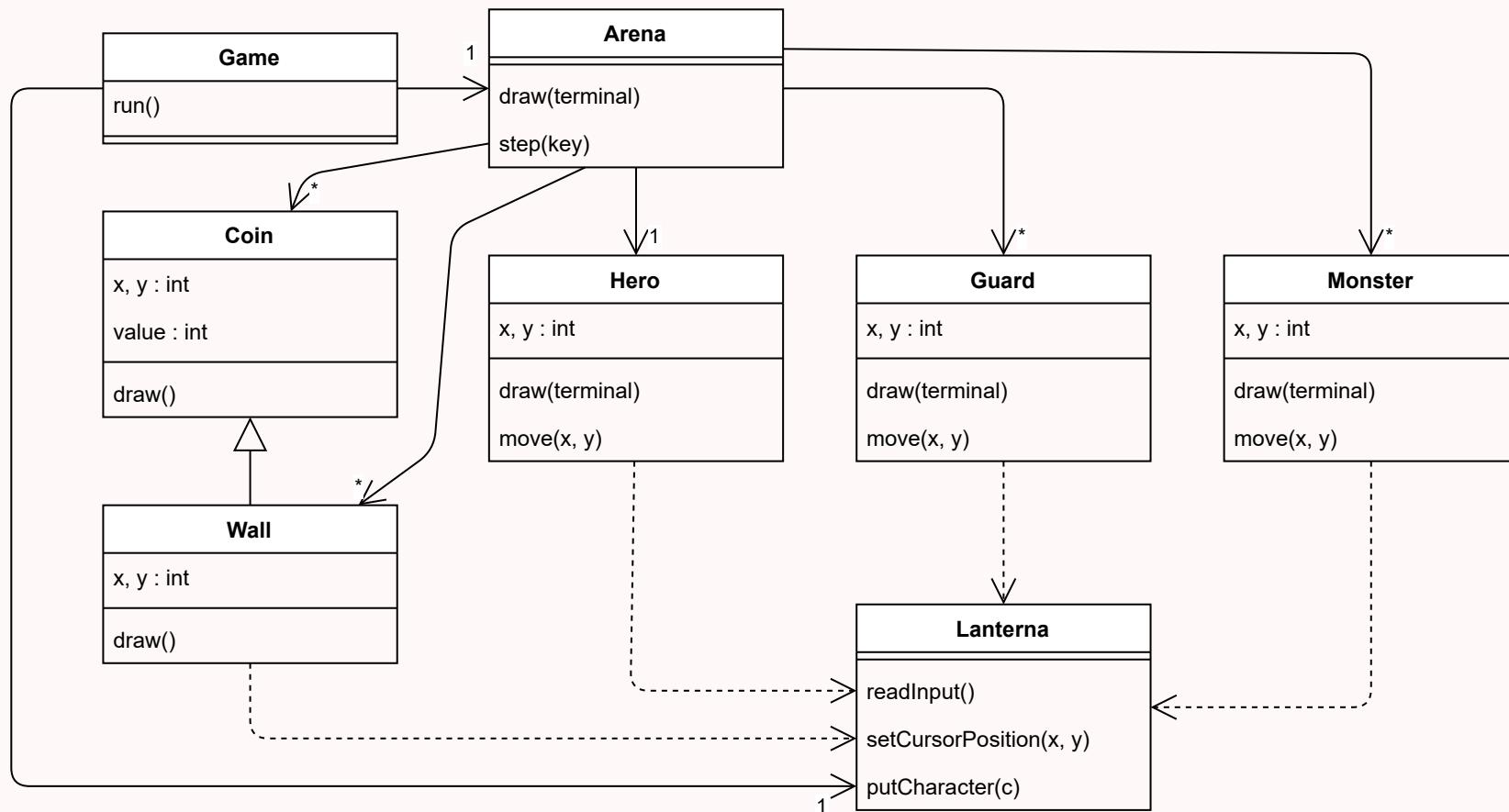
The Stable Abstractions Principle (SAP)

Stable packages should be abstract packages.

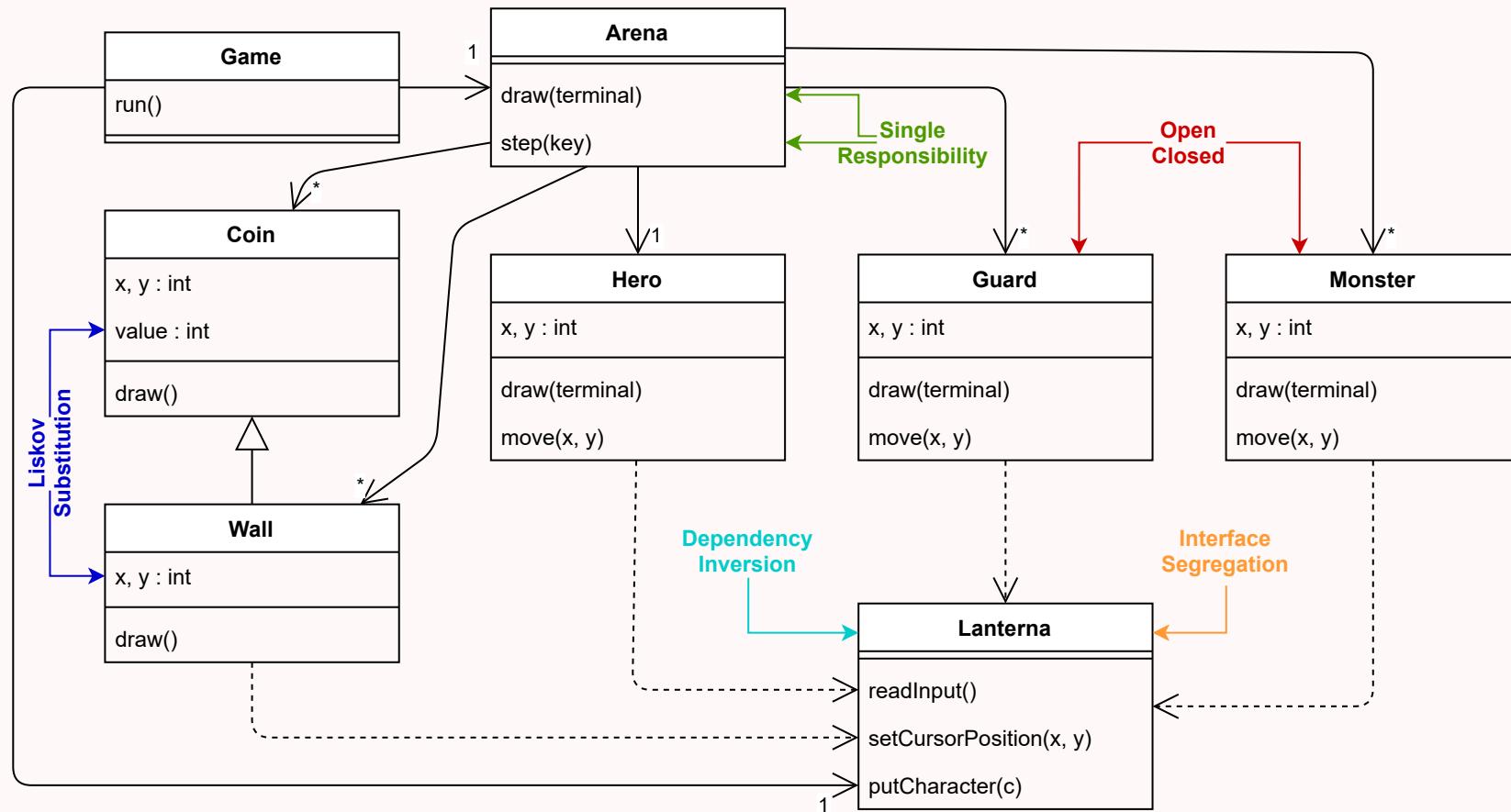
- A package can be said to be "harder to change" as more packages depend on it.
- So it should be made **abstract** so that it can be **extended** when necessary.
- A package that is **not used** by other packages can be "changed easily", so it can remain **concrete**.

An Example

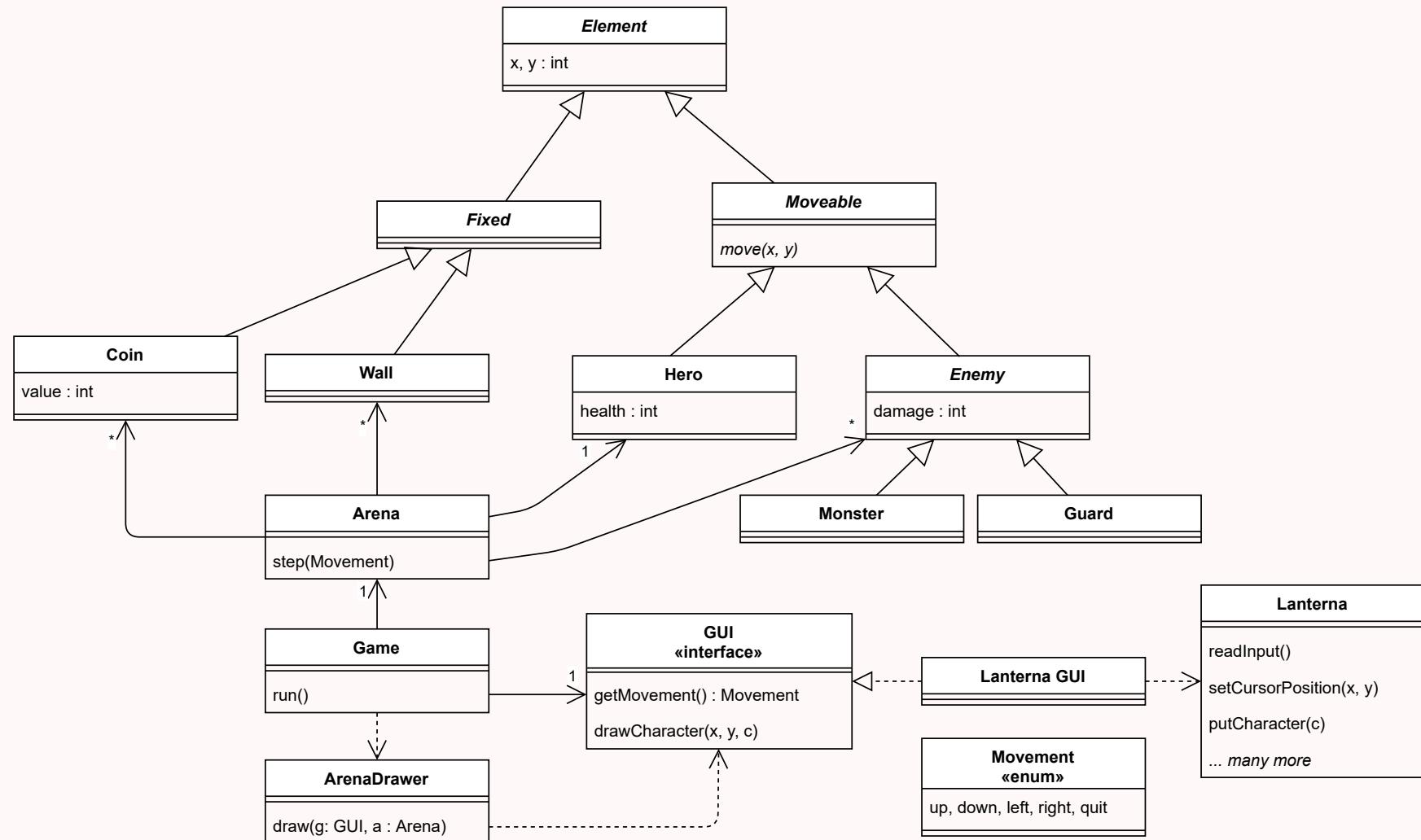
Bad Design



Violated Principles



Solid Design



UML

Class Diagrams

André Restivo

Index

Introduction

Classes

Inheritance

Associations

Interfaces

Aggregation

Dependency

Introduction

Types of Diagrams

In UML, there are two basic categories of diagrams:

- **Structure** diagrams show the static structure of the system being modeled: *class*, *component*, *deployment*, *object* diagrams, ...
- **Behavioral** diagrams show the dynamic behavior between the objects in the system: *activity*, *use case*, *communication*, *state machine*, *sequence* diagrams, ...

Class Diagrams

Class diagrams show the **classes** of the system, their **relationships** (including inheritance, aggregation, and association), and the **operations** and **attributes** of the classes.

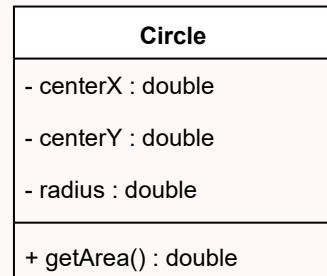
Class diagrams are used for different purposes:

- Conceptual **domain** modeling:
 - Illustrates meaningful conceptual classes in problem domain.
 - Represents real world concepts, not software components.
- Detailed **design** modeling:
 - Represents the concrete software components.

Classes

Class

The UML representation of a class is a **rectangle** containing **three compartments** stacked vertically:



Class Attribute List

The **middle** compartment lists each of the **attributes** of the class on a separate line.

Each line uses the following format:

 *name : attribute type*

For example:

 *width : double*

Attribute Default Value

Default values can be specified (optionally) in the attribute list section by using the following notation:

name : attribute type = default value

For example:

width : double = 0

Class Operations List

The **lowest** compartment lists each of the **operations** of the class on a separate line.

Each line uses the following format:

name(parameter list) : type of value returned

For example:

setRadius(radius : double) : void

Operation Parameters

When an **operation** has **parameters**, they are put inside **parentheses**.

Each parameter uses the **format**:

`parameter name : parameter type`

They can also have a **optional** "in" or "out" marking specifying if the parameter is an **input** or **output** parameter.

For example:

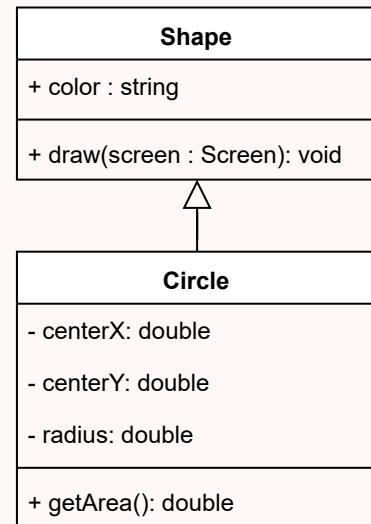
`setRadius(in radius : double) : void`

Inheritance

Inheritance

The ability of one class (child class) to **inherit** the identical **functionality** of another class (super class), and then **add new functionality** of its own.

Inheritance is indicated by a **solid line with a closed, unfilled arrowhead pointing at the super class**.

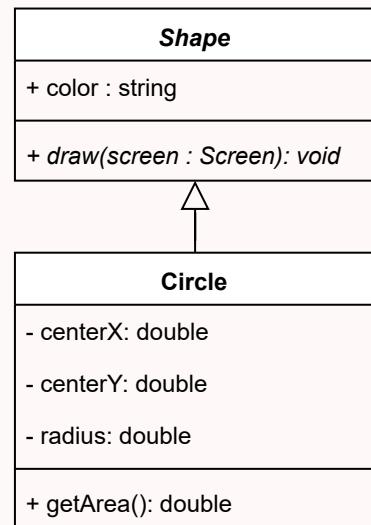


Abstract classes and operations

Abstract operations are operations where the class only provides the operation **signature** and not it's code.

Abstract classes are classes that contain abstract operations and, therefore, cannot be instantiated.

They are both represented in italic.

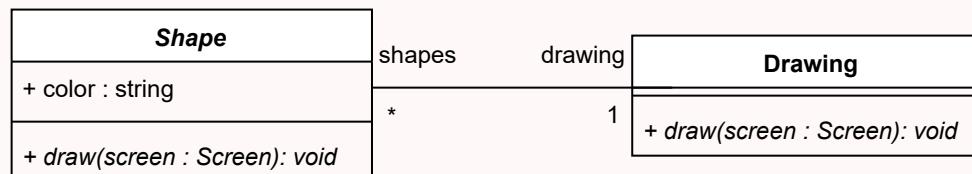


Associations

Bi-directional association

Associations are assumed to be **bi-directional** by default. This means that both classes are aware of each other.

A bi-directional association is indicated by a **solid line** between the two classes.



At either end of the line, you place a **role name** and a **multiplicity value**.

Multiplicity

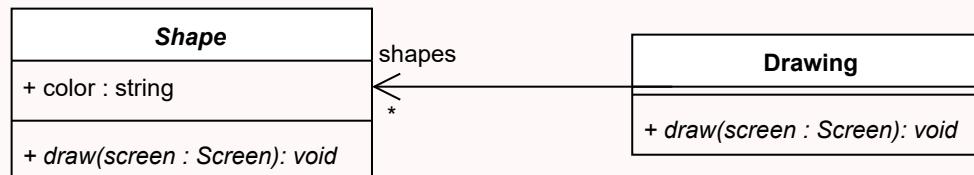
Some examples of possible multiplicities:

| Multiplicity | Shorthand | Cardinality |
|--------------|-----------|---|
| 0..0 | 0 | Collection must be empty |
| 0..1 | | No instances or one instance |
| 1..1 | 1 | Exactly one instance |
| 0..* | * | Zero or more instances |
| 1..* | | At least one instance |
| 5..5 | 5 | Exactly 5 instances |
| m..n | | At least m but no more than n instances |

Uni-directional association

In a uni-directional association, two classes are related, but only one class knows that the relationship exists.

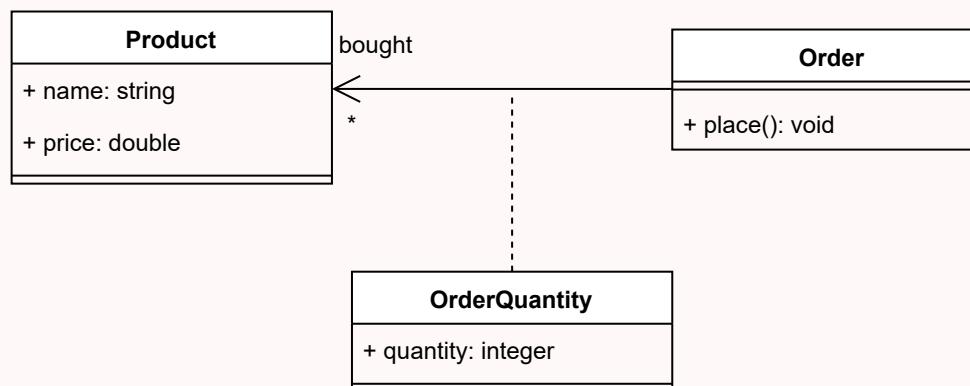
A uni-directional association is drawn as a solid line with an open arrowhead pointing to the known class.



Association Class

An association class includes information about a relationship.

It is represented like a normal class but has a dotted line connecting it to the association.

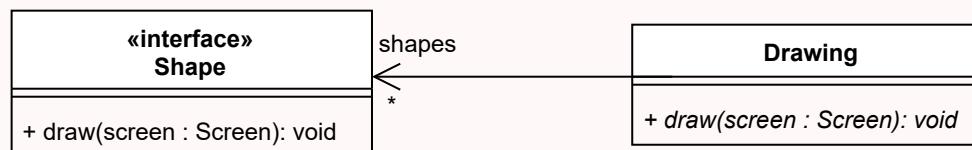


Interfaces

Interface

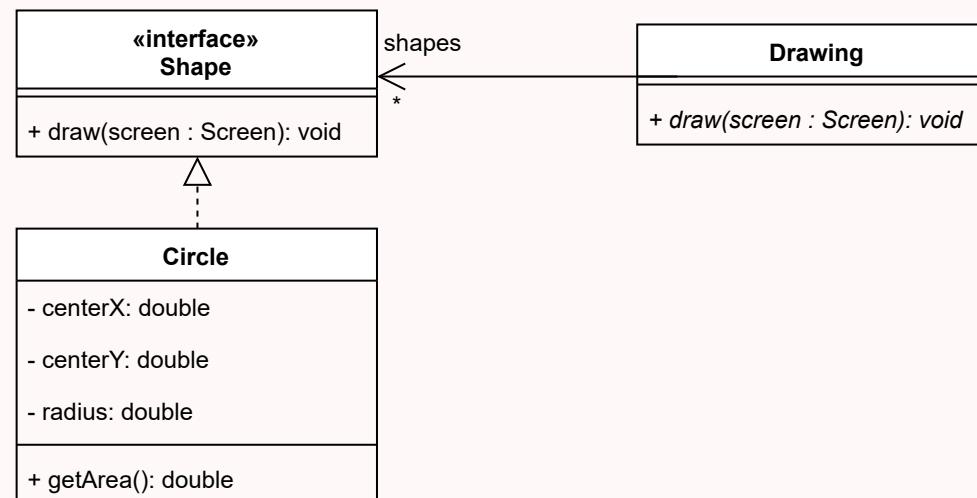
- An interface is a **description of the actions** that an object **can do**.
- The **combination of all public methods and properties** of an object.
- Interfaces can also be seen as **contracts** that other classes must fulfil.

In UML, an interface is depicted just like a class but with a **«interface» keyword**.



Implementation

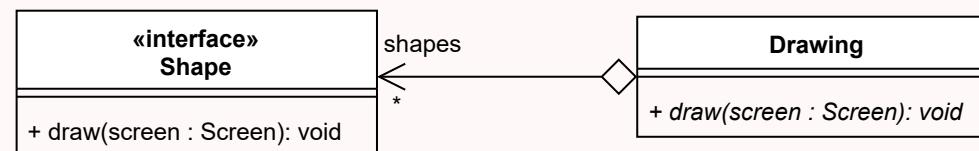
A class can **declare** that it **implements** a certain **interface** in a very similar way to inheritance (but with the line dotted and not solid).



Aggregation

Aggregation

- Aggregation is a special type of **association** used to model a "whole to its parts" relationship.
- An association with an **aggregation relationship** indicates that **one class is a part of another class**.
- In an aggregation relationship, the **child class instance can outlive its parent class**.
- To represent an aggregation we use an **unfilled diamond shape** on the **parent's association end**.



Composition

- The **composition aggregation** relationship is another, **stronger**, form of the aggregation relationship.
- In a **composition aggregation** relationship, the **child** class instance **cannot outlive its parent class**.
- To represent a composition aggregation we use an **filled diamond shape** on the parent's association end.

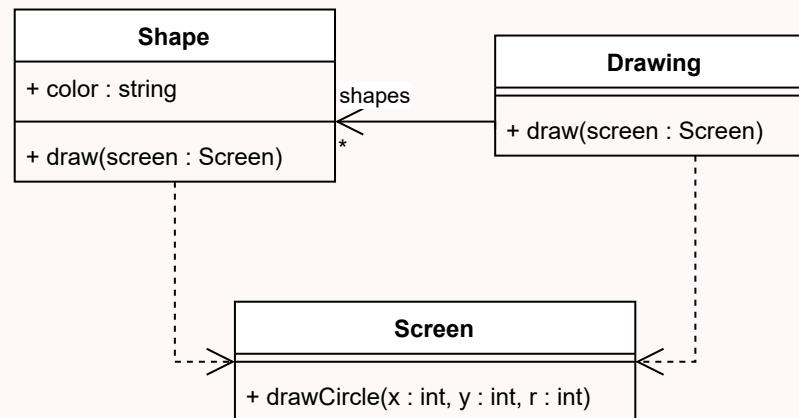


Dependency

Dependency

Represents a **dependency** between two elements of a UML diagram (e.g., classes).

Important when we want to show that **changes** to an element **may impact** another one; even when there is no association (as in an attribute referencing the other class) between them.



Design Patterns

André Restivo

Index

Introduction

Factory Method

Composite

Command

Observer

Strategy

State

Adapter

Decorator

Singleton

Abstract Factory

Architectural Patterns

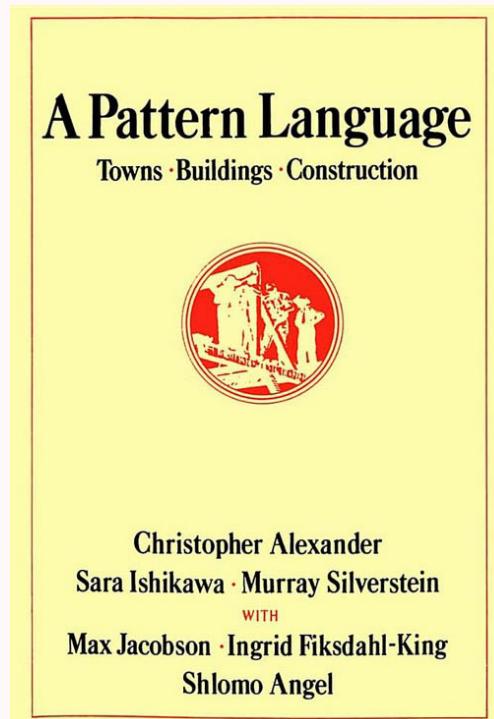
Reference

- Gamma, Erich, et al. "Design Patterns: Elements of Reusable Object-Oriented Software." (1994).
- [Source Making](#)
- [Refactoring Guru](#)

Introduction

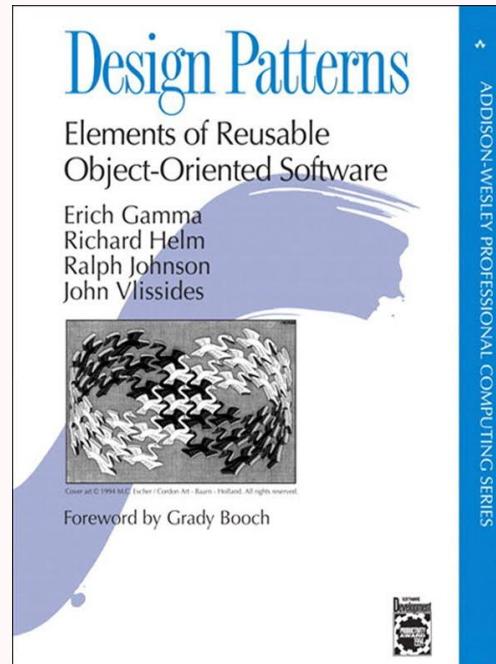
Patterns

Patterns originated as an architectural concept by Christopher Alexander (1977/78).



(Software) Design Patterns

Design patterns gained popularity in computer science after the book "**Design Patterns: Elements of Reusable Object-Oriented Software**" was published in **1994** by the so-called "**Gang of Four**".



Design Pattern

A **general, reusable** solution to a **commonly occurring** problem within a given **context** in software design.

It is **not a finished design** that can be transformed directly into source code.

It is a **description or template** for how to **solve** a problem that can be **used in many different situations**.

GoF Patterns

The twenty-three design patterns described by the Gang of Four:

| Creational | Structural | Behavioral |
|------------------|------------|-------------------------|
| Abstract Factory | Adapter | Chain of Responsibility |
| Builder | Bridge | Command |
| Factory Method | Composite | Interpreter |
| Prototype | Decorator | Iterator |
| Singleton | Facade | Mediator |
| | Flyweight | Memento |
| | Proxy | Observer |
| | | State |
| | | Strategy |
| | | Template Method |
| | | Visitor |

Documentation

The documentation for a design pattern describes the context in which the pattern is used, the forces within the context that the pattern seeks to resolve, and the suggested solution.

| | |
|----------------------|-----------------------|
| Pattern Name | Classification |
| Intent | Collaboration |
| Also Known As | Consequences |
| Motivation | Implementation |
| Applicability | Sample Code |
| Structure | Known Uses |
| Participants | Related Patterns |

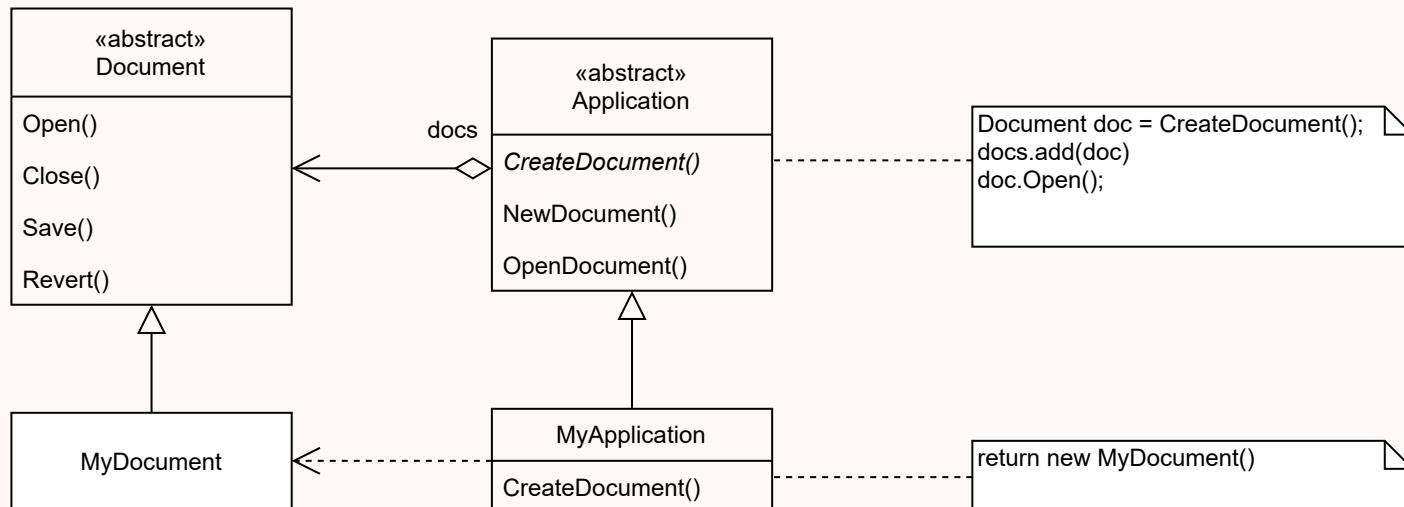
Factory Method

Factory Method

"Define an interface for creating an object, but let sub-classes decide which class to instantiate."

Motivation

A framework for applications that can present multiple documents to the user.



Applicability

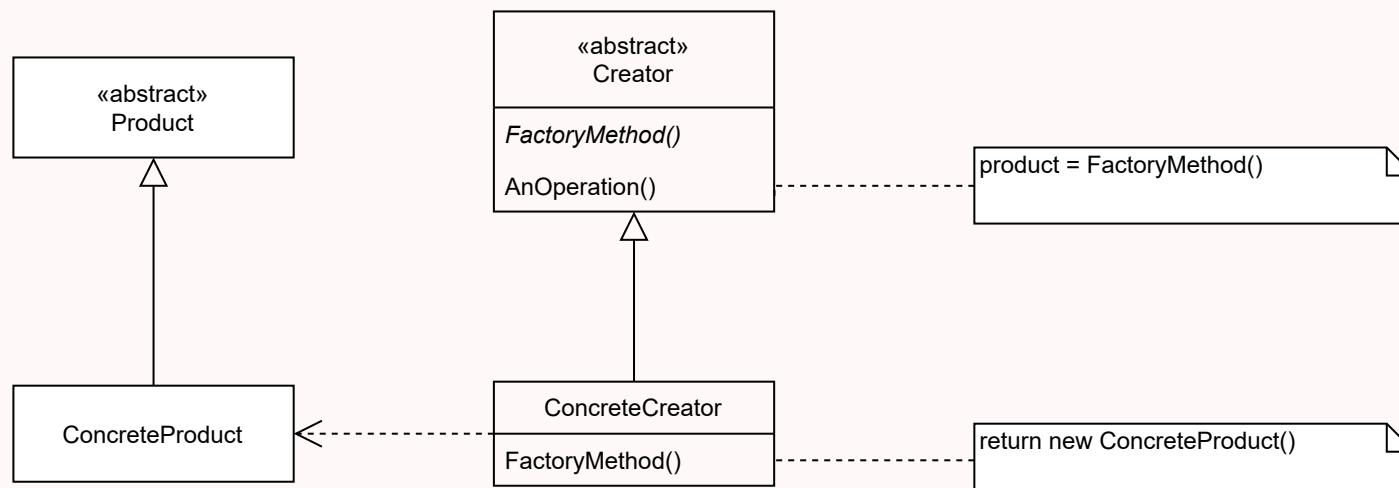
Use the **factory method** pattern when:

- a class can't anticipate the class of objects it must create.
- a class wants the subclasses to specify the objects it creates.
- classes delegate responsibility to one of several helper classes, and you want to localize the knowledge of which helper subclass is the delegate.

Consequences

Factory methods eliminate the need to bind application-specific classes into your code. The code only needs to deal with the Product *interface*; therefore it can work with any user-defined ConcreteProduct classes.

Structure



Variations

- Creator might not be abstract and provide a default implementation for the FactoryMethod.
- Factory Method might take a parameter specifying the type of product to create.
- Using Generics/Templates to avoid *subclassing* the Creator.

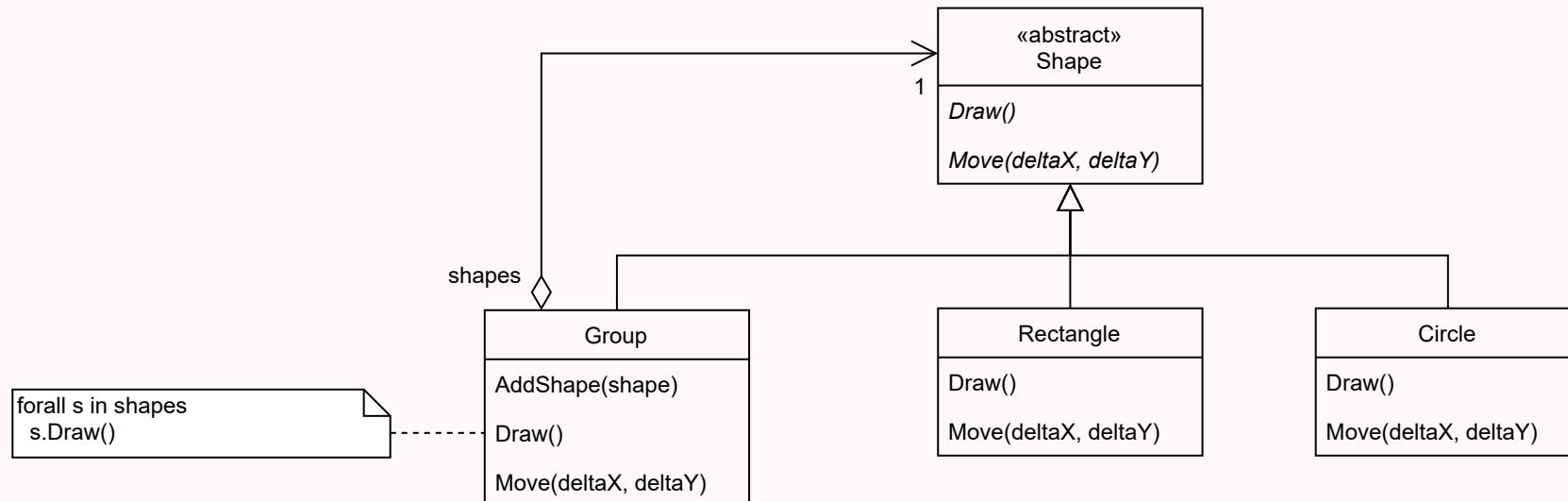
Composite

Composite

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

Motivation

A graphics application where shapes can be composed into groups.



Applicability

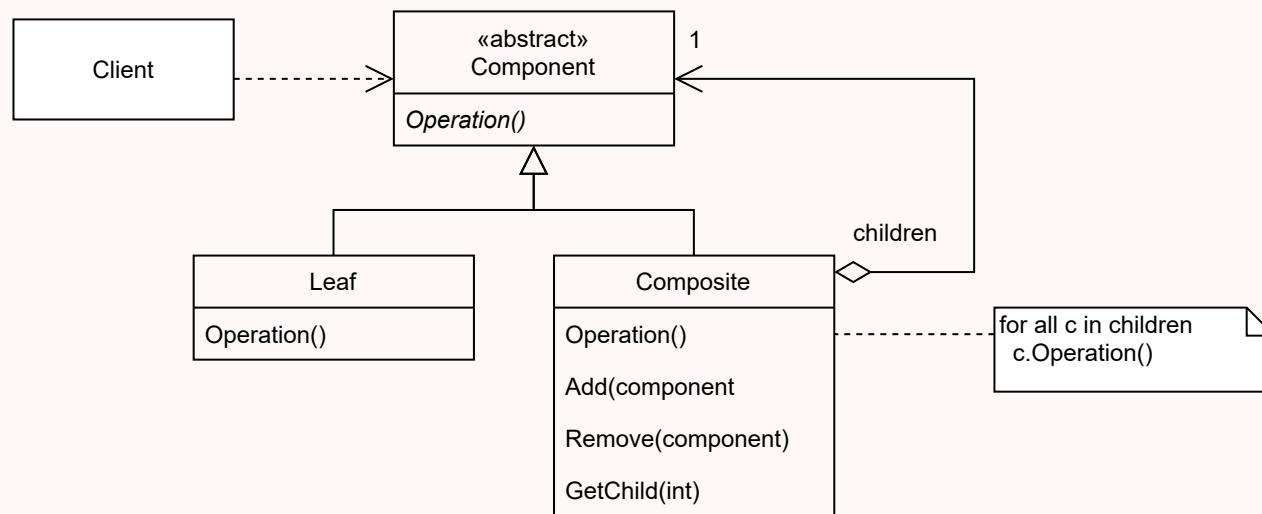
Use the **composite** pattern when:

- you want to represent part-whole hierarchies of objects.
- you want clients to be able to ignore the difference between compositions of objects and individual objects.

Consequences

- Primitive objects can be composed into more complex objects.
- Clients can be kept simple.
- Easier to add new types of components.

Structure



Variations

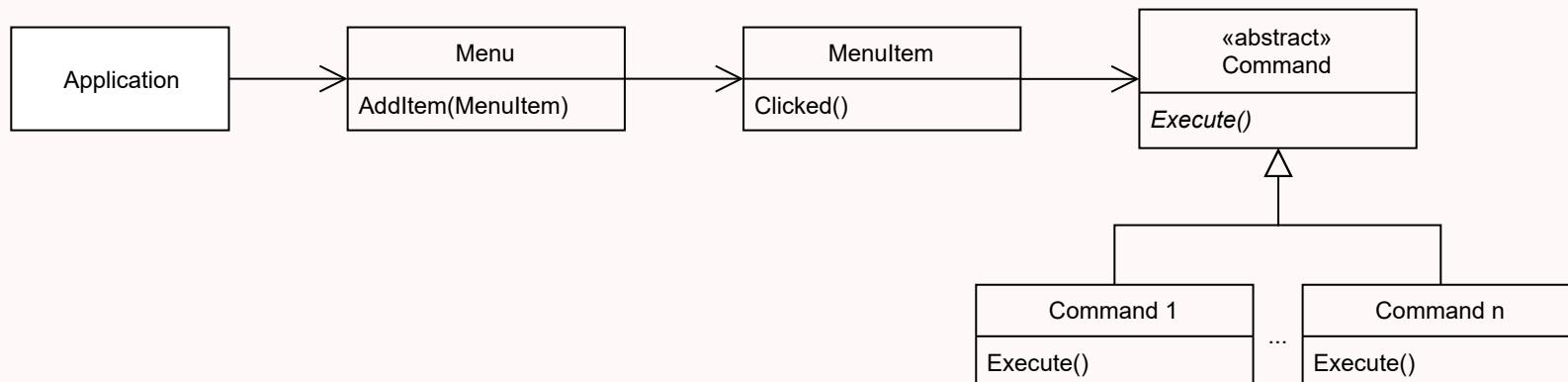
- Maintaining **references** from **child** components to their **parents**.
- **Sharing** components.
- **Child ordering**.
- **Caching** to improve performance.

Command

Command

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

Motivation



Applicability

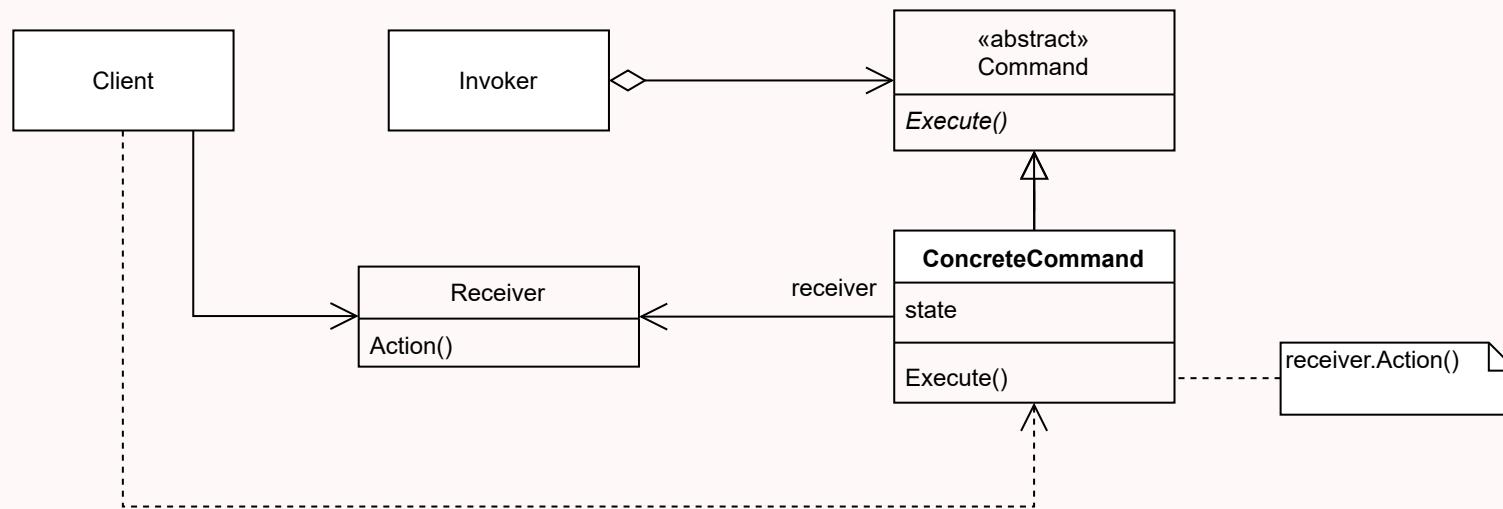
Use the **command** pattern when:

- **parameterize** objects by an action to perform.
- **specify, queue, and execute** requests at different times.
- support **undo/redo** operations.
- support **logging** changes so they can be reapplied.
- **structure** a system around **high-level** operations built on **primitive** operations.

Consequences

- Decouples the object that invokes the operation from the one that knows how to perform it.
- Commands can be extended and manipulated like any other object.
- You can create **Composite** commands.
- It's easy to add new commands.

Structure



Variations

- Commands only **delegating** to Receiver actions or doing **all the work** by themselves.
- Support **undo/redo** instead of only action.
- Avoiding **error accumulation** in undo operations.

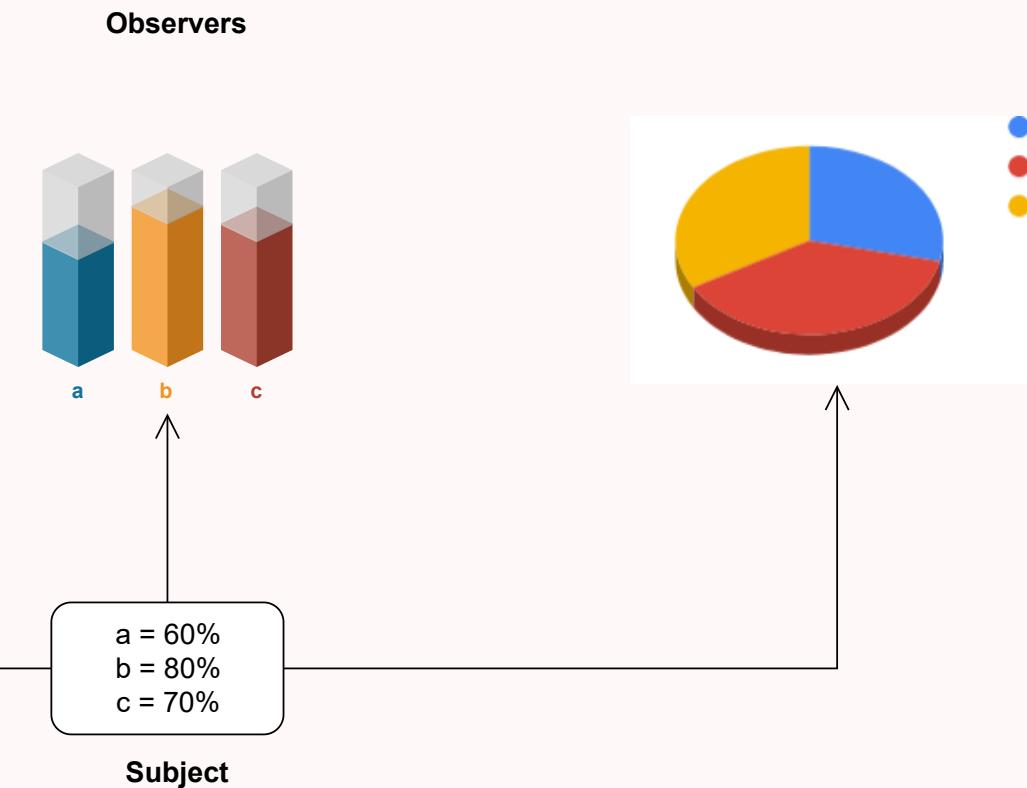
Observer

Observer

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

Motivation

| a | b | c |
|-----------|-----------|-----------|
| 30 | 40 | 10 |
| 60 | 80 | 70 |
| 40 | 40 | 40 |
| 10 | 20 | 70 |



Applicability

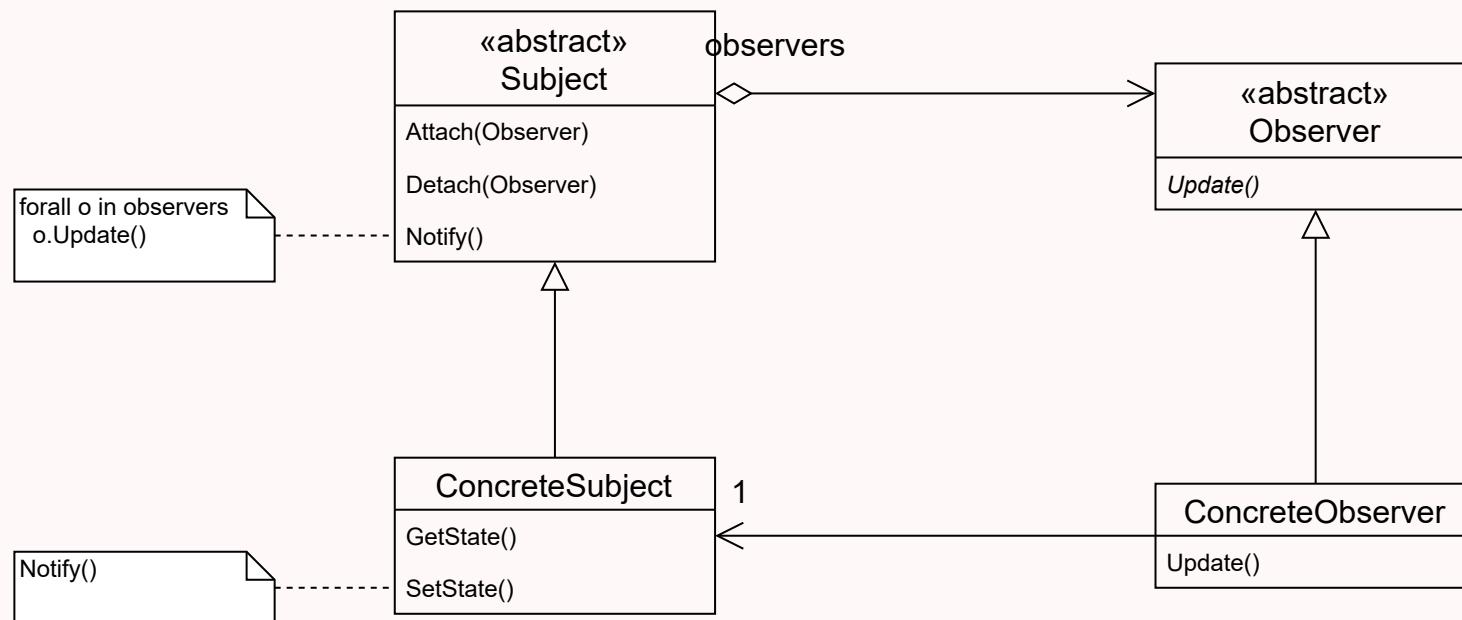
Use the **observer** pattern when:

- When an abstraction has two aspects one dependent on the other.
- When a change to one object requires changing others.
- When an object should be able to notify other objects without making assumptions about who those objects are.

Consequences

- Abstract coupling between subject and observer.
- Support for broadcast communication.
- Unexpected updates.

Structure



Variations

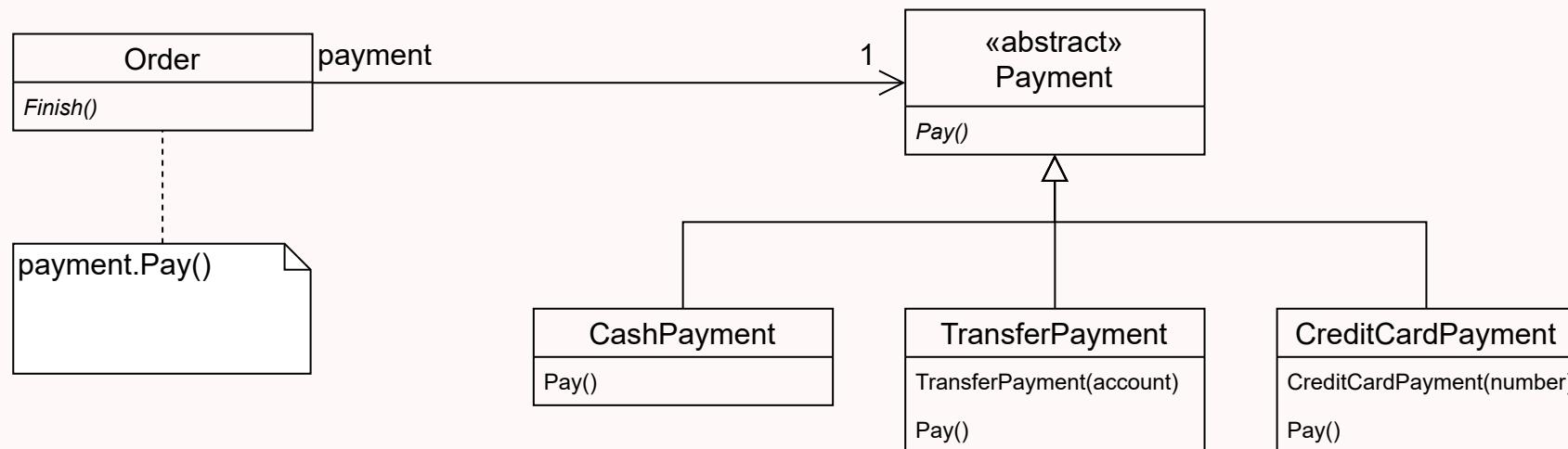
- Observing more than one subject.
- Who triggers the update (client or subject)?
- Push and pull models.
- Specifying "events of interest" explicitly.

Strategy

Strategy

"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary depending from clients that use it."

Motivation



Applicability

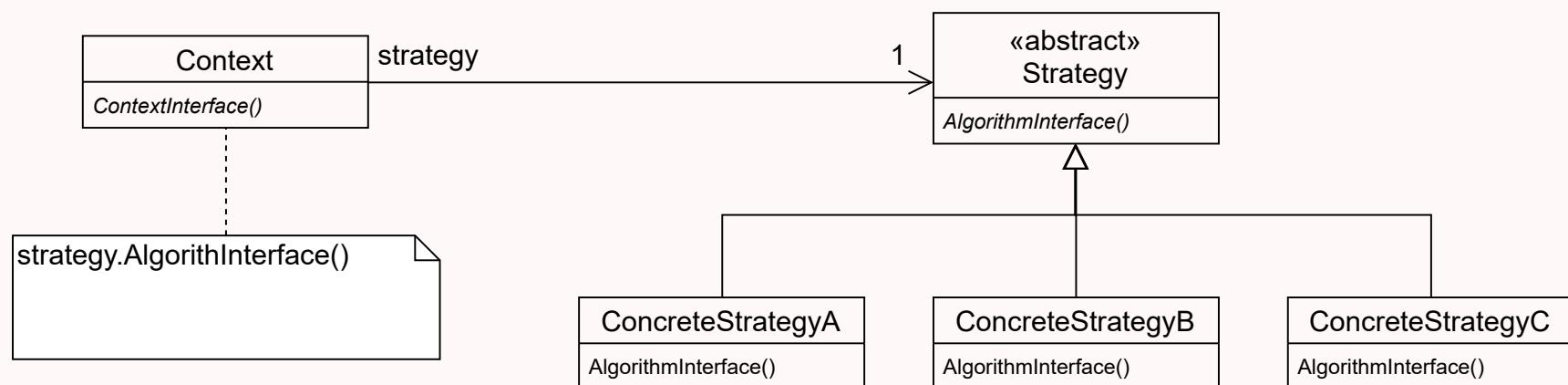
Use the **strategy** pattern when:

- many related classes differ only in their behavior.
- you need different variants of an algorithm.
- an algorithm uses data that clients should not know about.
- a class defines many behaviors that appear in multiple conditional statements.

Consequences

- An alternative to subclassing.
- Eliminates conditional statements.
- Provides different implementations.
- Clients must be aware of different strategies.

Structure



Variations

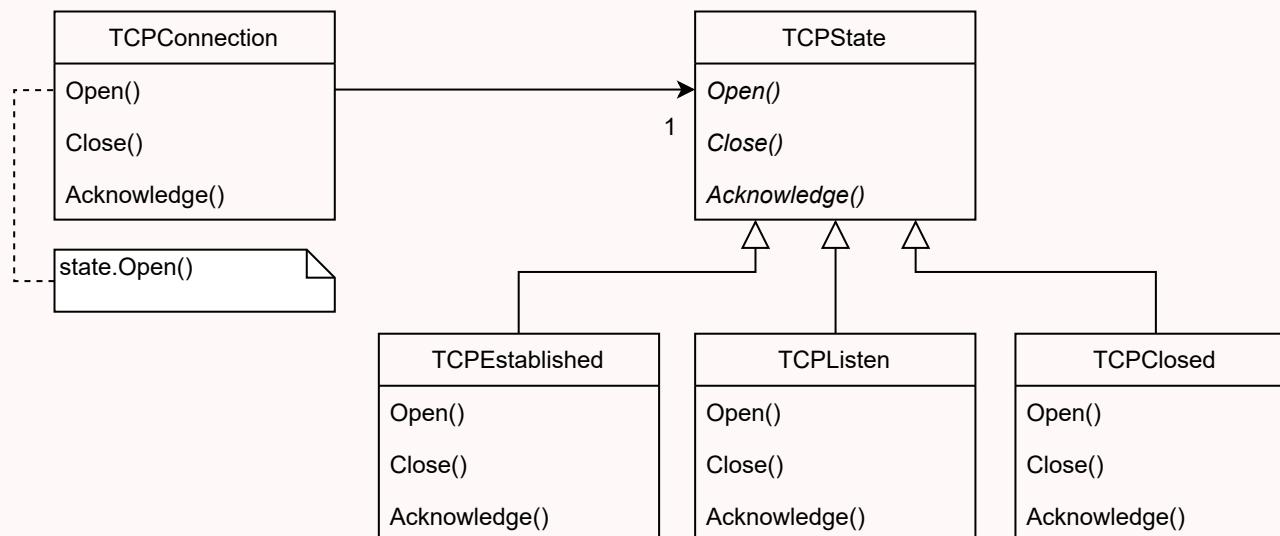
- Strategy and Context must have well defined interfaces for exchanging any needed data.
- Strategy can be optional if context has default behavior.

State

State

"Allow an object to alter its behavior when its internal state changes. The object will appear to change its class."

Motivation



Applicability

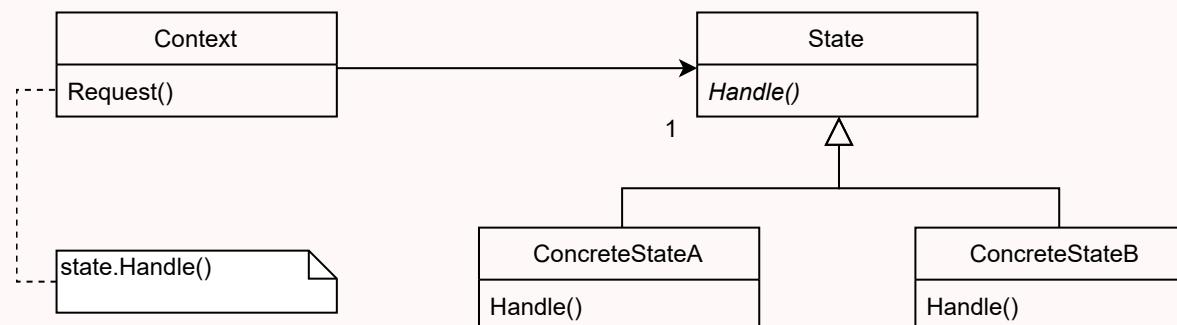
Use the **state** pattern when:

- an object behavior depends on its state, and it must change that state in run-time.
- operations have large, multipart conditional statements that depend on one or more enumerated constants.

Consequences

- Localizes and partitions behavior for different states.
- Makes state transitions explicit.

Structure



Variations

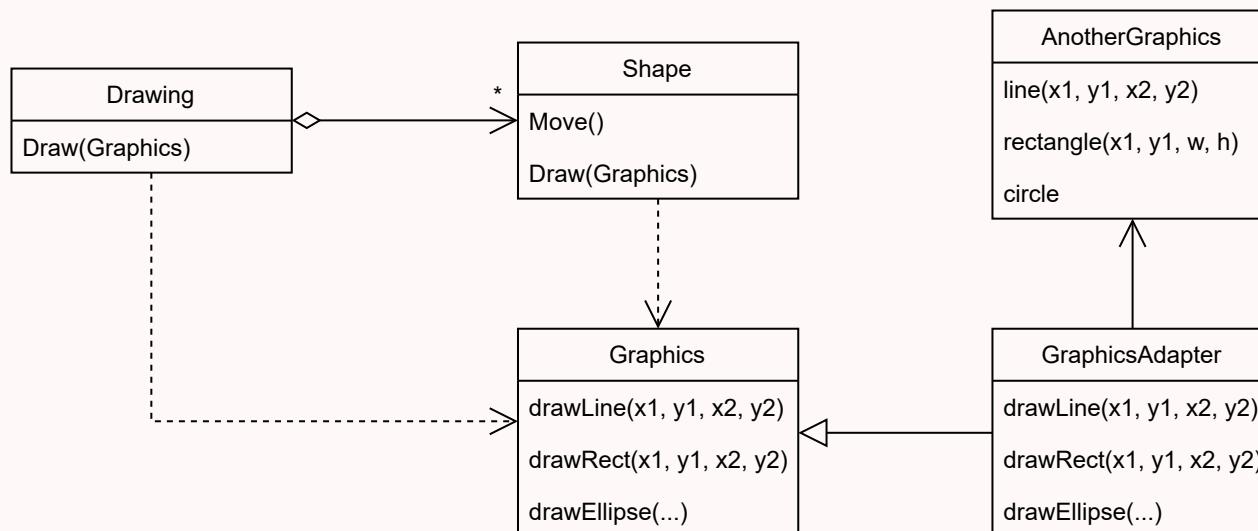
- Who defines the criteria for state transitions? More flexible solution is to let the states define the transitions.
- When are state objects created and destroyed? Easier to create and destroy when state changes. Better to only create them once and never destroy them.

Adapter

Adapter

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

Motivation

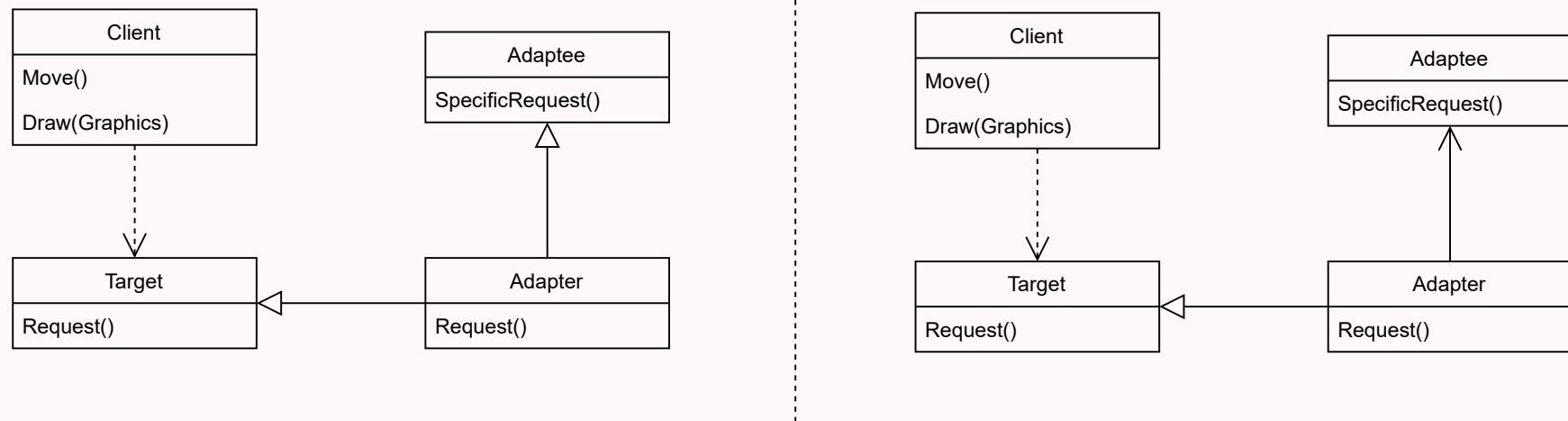


Applicability

Use the **adapter** pattern when:

- you want to use an existing class, and the interface does not match the one you need.
- you want to create a reusable class that work with unforseen classes.

Structure



Two different alternatives:

- **Class adapter:** Using multiple inheritance (if available).
- **Object adapter:** Using composition.

Decorator

Decorator

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

Motivation



Applicability

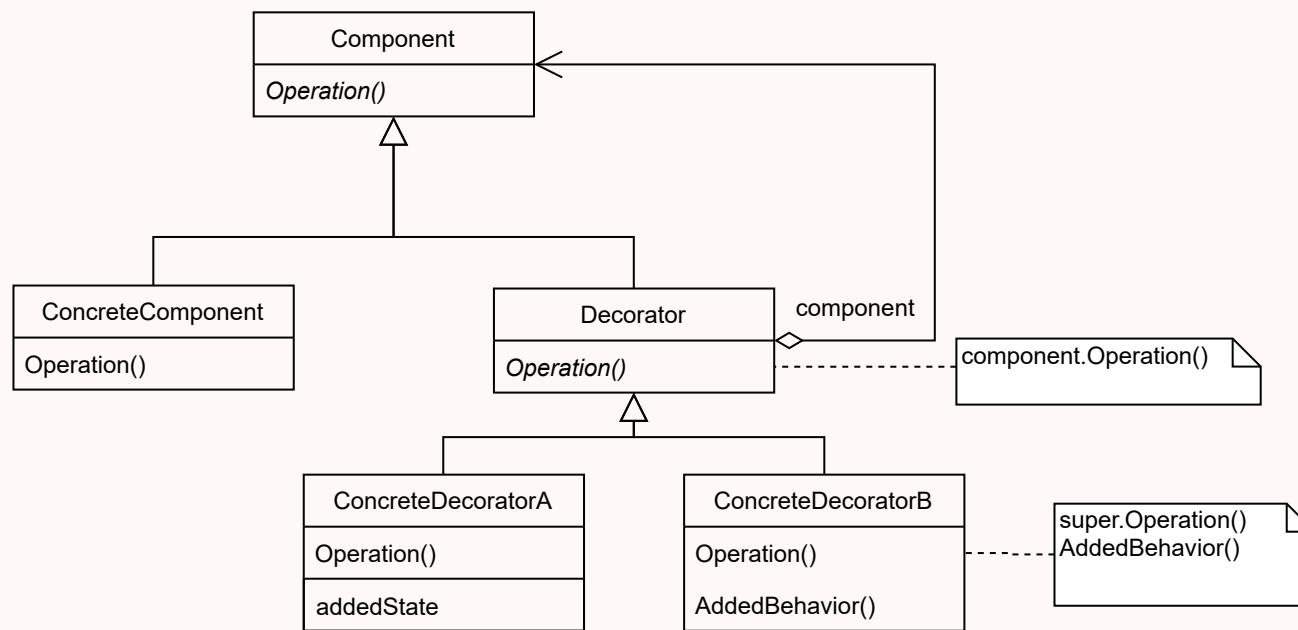
Use the **decorator** pattern when:

- to add responsibilities for **individual** objects dynamically and transparently.
- for responsibilities that can be withdrawn.
- when extension by subclassing is impractical (many different combinations → explosion of subclasses).

Consequences

- More flexible than static inheritance.
- Avoids classes with too many features and responsibilities.
- Lots of little objects.

Structure



"*Changing the object guts (Strategy) versus changing the object skin (Decorator)*".

Singleton

Singleton

"Ensure a class only has one instance and provide a global point to access it."

Applicability

Use the **singleton** pattern when:

- there must be exactly one instance of a class.
- when the sole instance must be extensible by subclassing.

Consequences

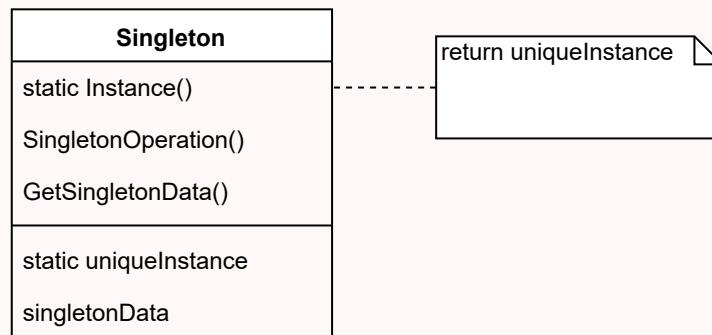
The **singleton** pattern is considered an **anti-pattern**:

- The assumption that there will ever be only one instance is often broken during a project's lifetime.
- Makes it very difficult to test code.
- Difficult to implement correctly when taking multi-threading into account.

What to use instead

- Instantiate a single instance and propagate it to places that use the object as a parameter.

Structure



Variations

When to create the unique instance?

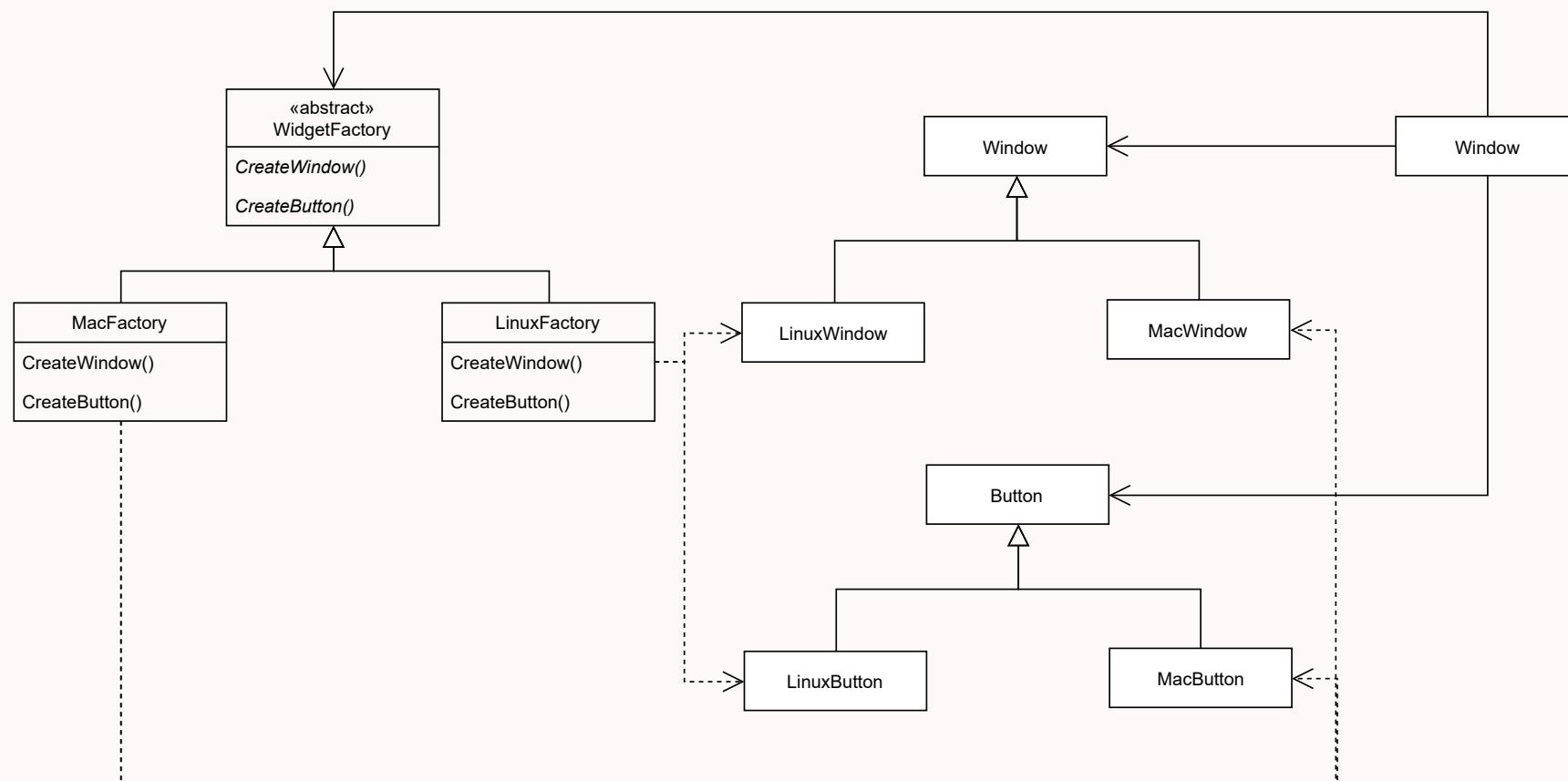
- At application start.
- When `instance()` is called for the first time.

Abstract-Factory

Abstract-Factory

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

Motivation



Applicability

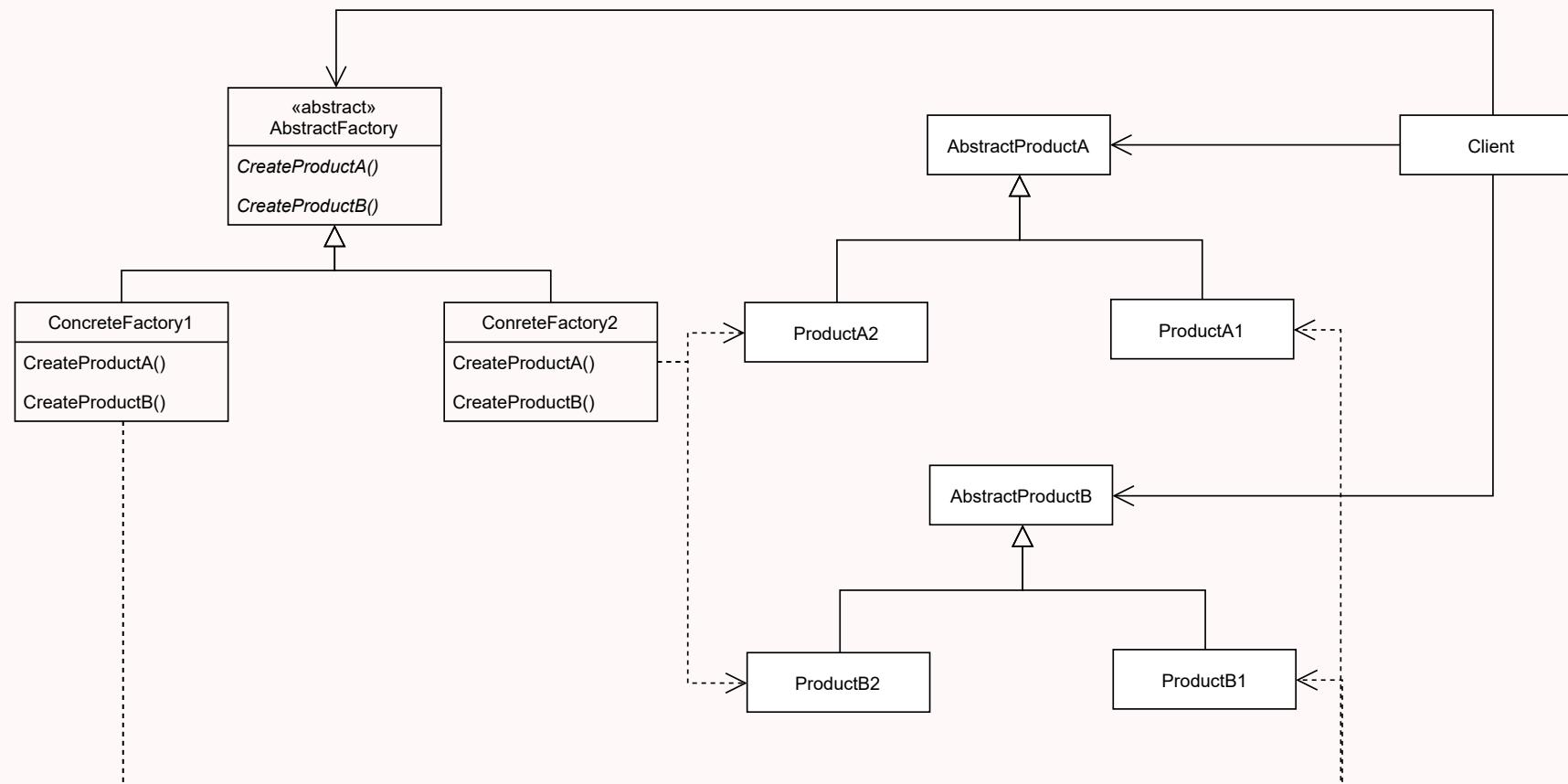
Use the **abstract factory** pattern when:

- a system should be independent from how its products are created, composed and represented.
- a system should be configurable with one or more families of products.
- a family of products is designed to work together and you need to reinforce this constraint.

Consequences

- It isolates concrete classes.
- It makes exchanging product families easy.
- It promotes consistency among products.
- Supporting new types of products is difficult.

Structure



Architectural Patterns

Architectural Patterns

An **Architectural style** is responsible for how we should organize our code (monolithic, pipes and filters, plugins, microservices, ...).

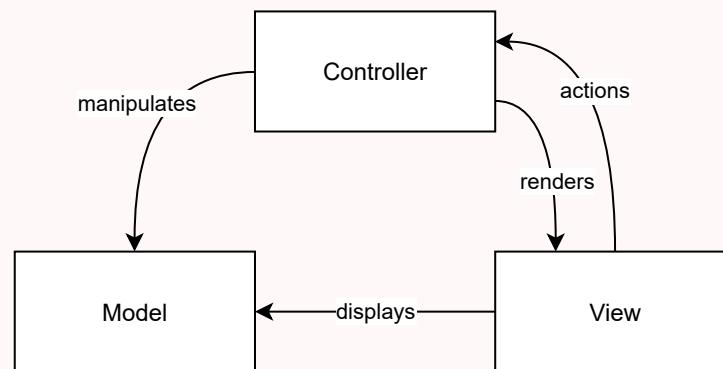
Architectural patterns are patterns that deal with the architectural style of software.

Some examples:

- Model-View-Controller (commonly used in GUIs)
- Pipe-Filter (pipes guide data from filter to filter)
- Broker (message queues e.g. RabbitMQ)
- ...

Model-View-Controller (MVC)

An architectural pattern commonly used for developing user interfaces that divides an application into **three** parts.



- The **model** only represents the **data**.
- The **view displays** the **model** data, and sends user **actions** to the **controller**.
- The **controller** provides **model** data to the **view**, and **interprets** user **actions**.

MVC Variants

There are several variants to the MVC pattern and even the MVC hasn't got a single interpretation:

- **HMVC** - Hierarchical Model-View-Controller: Each visual component has its own MVC model.
- **MVVM** - Model-View-ViewModel: Uses a ViewModel as a binder between the model and the view.
- **MVP** - Model-View-Presenter: The Presenter sits in the middle of the View and the Model mediating the actions between them.

Refactoring

André Restivo

Index

Introduction

Code Smells

Refactoring

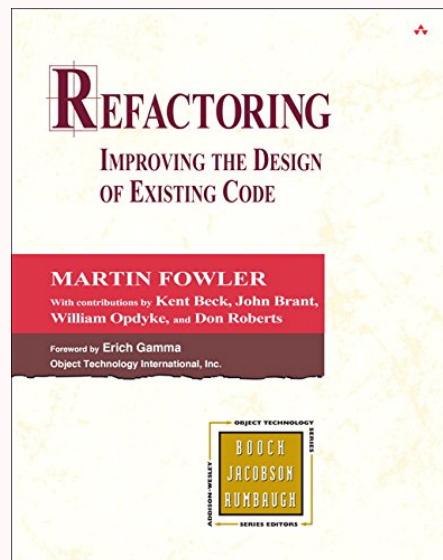
Reference

- Fowler, Martin. **Refactoring: Improving the design of existing code**. Addison-Wesley Professional, 1999. And also the accompanying [website](#) and online [catalog](#).
- Kerievsky, Joshua. **Refactoring to Patterns**. Pearson Deutschland GmbH, 2005.
- [Refactoring Guru](#)

Introduction

Refactoring

- Refactoring is a **controlled** technique for **improving** the design of an **existing** code base.
- Its essence is applying a series of **small** behavior-preserving transformations, each of which "*too small to be worth doing*".
- However the **cumulative** effect of each of these transformations is quite **significant**.



Two Hats

 *A metaphor by Kent Beck*

When you are developing, you divide your time into **two distinct activities**:

- When you are **adding** functionalities, you **shouldn't** be **changing** code.
- When you are **refactoring**, you **shouldn't** be adding new capabilities.

Why Refactor?

- It **improves** the design of software. It prevents the design of software from decaying.
- It makes software easier to **understand**.
- It helps you find **bugs**. Refactoring forces you to think deeply about your code.
- It helps you program **faster**. A good design is essential to maintaining speed in software development.

Importance of Testing

- Refactoring is intended to improve **nonfunctional** attributes of the software.
- Having a good **testing suite** is of paramount importance **before** refactoring to ensure the code still behaves as expected.

Code Smells

Code Smells

- A code smell is a **surface indication** that usually corresponds to a **deeper problem** in the system.
- A code smell is something that's **quick to spot** (*sniffable*).
- A code smell **doesn't always** indicate a **problem**. Smells aren't inherently bad on their own, they are often an **indicator of a problem** rather than the problem themselves.

1. Bloaters

Code, methods and classes that are so large and complex that they are hard to work with.

- **Long Method** A method that contains too many lines of code.
- **Large Class** A class that contains many fields/methods/lines of code.
- **Primitive Obsession** Use of primitives instead of small objects for simple tasks.
- **Long Parameter List** More than three or four parameters for a method.
- **Data Clumps** Different parts of the code containing identical groups of variables.

2. Object-Orientation Abusers

Incorrect application of object-oriented programming.

- **Switch Statements** Complex switch/if operators.
- **Temporary Field** Temporary fields that get their values only under certain circumstances.
- **Refused Bequest** If a subclass uses only some of the methods and properties inherited from its parents.
- **Alternative Classes with Different Interfaces** Two classes perform identical functions but have different method names.

3. Change Preventers

Make changing the code harder.

- **Divergent Change** Changing many unrelated methods when you make changes to a class.
- **Shotgun Surgery** Many small changes to many different classes.
- **Parallel Inheritance Hierarchies** Whenever you create a subclass for a class, you find yourself needing to create a subclass for another class.

4. Dispensables

Things that would make the code cleaner if they didn't exist.

- **Comments** A method is filled with explanatory comments.
- **Duplicate Code** Two code fragments look almost identical.
- **Lazy Class** Classes that don't do much.
- **Data Class** Class that contains only fields and crude methods for accessing them.
- **Dead Code** A variable, parameter, field, method or class that is no longer used.
- **Speculative Generality** There's an unused class, method, field or parameter that was created to support anticipated future features

5. Couplers

Excessive coupling between classes.

- **Feature Envy** A method accesses the data of another object more than its own data.
- **Inappropriate Intimacy** One class uses the internal fields and methods of another class.
- **Message Chains** In code you see a series of calls resembling: `a->b()->c()->d()`.
- **Middle Man** If a class only delegates work to another class, why does it exist at all?

Refactoring

Some Examples

Categories

The *Refactoring* book by Martin Fowler, divides refactoring into 7 categories:

1. Composing Methods
2. Moving Features Between Objects
3. Organizing Data
4. Simplifying Conditional Expressions
5. Making Methods Calls Simpler
6. Dealing With Generalization
7. Big Refactorings

In total, the book presents 70 different refactorings.

Refactoring Structure

- **Name:** so we can build a vocabulary of refactorings.
- **Summary:** the situation in which you need the refactoring and what it does.
- **Motivation:** why the refactoring should be done (and when it shouldn't).
- **Mechanics:** concise instructions on how to do the refactoring.
- **Example:** simple and normally with before and after code.

Normally with code smells that are resolved using the refactoring and other related refactorings.

1. Composing Methods (I)

Streamlining methods, removing code duplication and making future improvements easier.

These ones deal with **classes** and **methods**:

- **Extract Method** You have a code fragment that can be grouped together.
- **Inline Method** When a method body is more obvious than the method itself.
- **Replace Method with Method Object** You have a long method that uses local variables in such a way that you cannot apply *Extract Method*.
- **Substitute Algorithm** You want to replace an algorithm with one that is clearer.

1. Composing Methods (II)

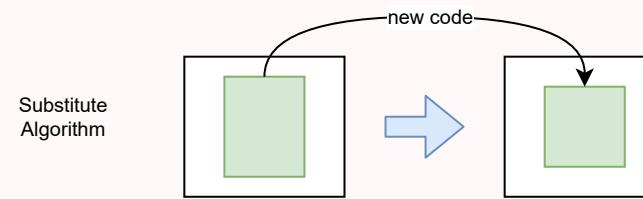
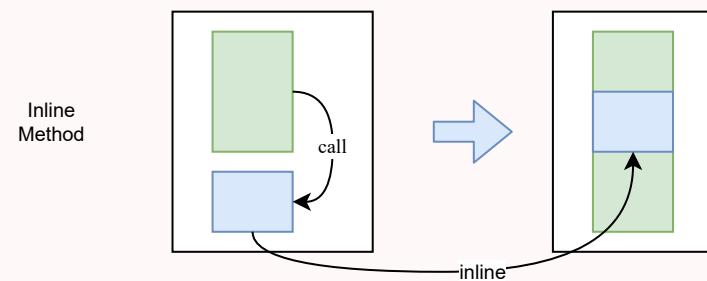
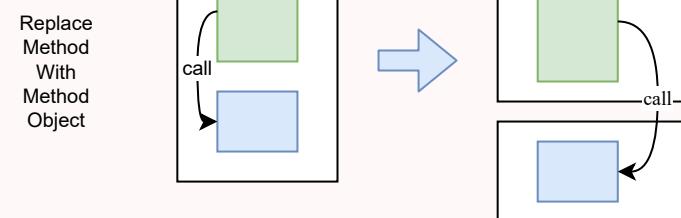
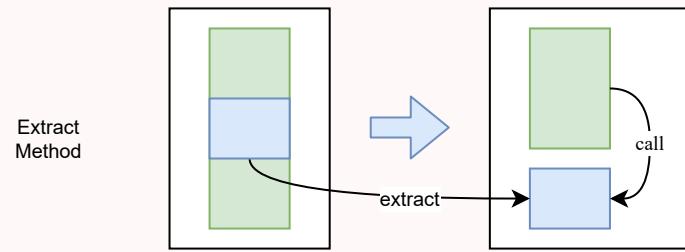
Streamlining methods, removing code duplication and making future improvements easier.

These ones deal with **methods** and **temporary variables**:

- **Inline Temp** You have a temporary variable that is assigned the result of a simple expression and nothing more.
- **Extract Variable** You have an expression that is hard to understand.
- **Split Temp Variable** You have a local variable that is used to store various intermediate values inside a method.
- **Replace Temp with Query** You place the result of an expression in a local variable for later use in your code.
- **Remove Assignments to Parameters** Some value is assigned to a parameter inside method's body.

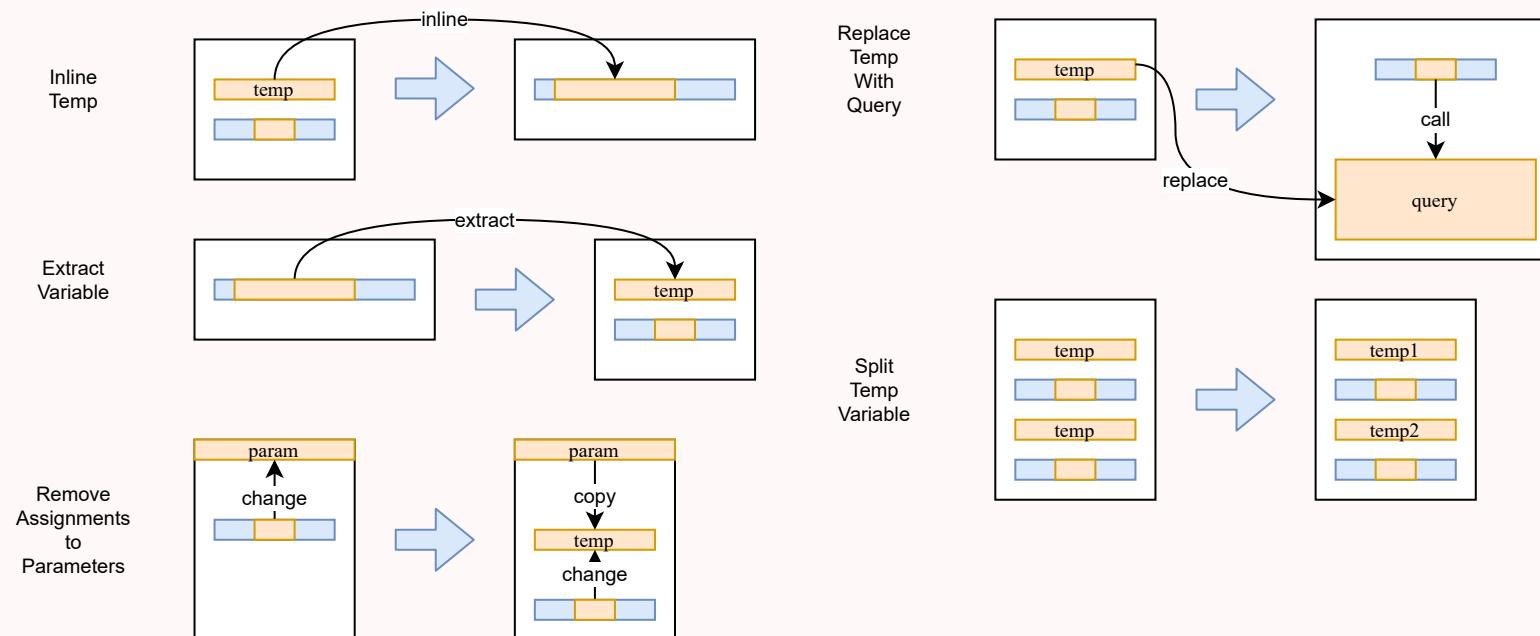
1. Composing Methods (III)

These ones deal with **classes** and **methods**.



1. Composing Methods (IV)

These ones deal with **methods** and temporary **variables**.



1. Composing Methods: Extract Method

```
    () {  
printBanner();  
  
    // Print details.  
    System.out.println("name: " + name);  
    System.out.println("amount: " + getOutstanding());  
}
```

1. Composing Methods: Extract Method

```
    () {  
printBanner();  
  
// Print details.  
System.out.println("name: " + name);  
System.out.println("amount: " + getOutstanding());  
}
```

Refactored into:

```
    () {  
printBanner();  
printDetails(getOutstanding());  
}  
  
    (      outstanding) {  
System.out.println("name: " + name);  
System.out.println("amount: " + outstanding);  
}
```

Can be used to eliminate: **Duplicate Code, Long Method, Feature Envy, Switch Statements, Message Chains, Comments and Data Class.**

1. Composing Methods: Inline Method

```
// ...
{
    () {
        moreThanFiveLateDeliveries() ? 2 : 1;
    }
    () {
        number0fLateDeliveries > 5;
    }
}
```

1. Composing Methods: Inline Method

```
// ...
{
    () {
        moreThanFiveLateDeliveries() ? 2 : 1;
    }
    () {
        numberOfRowsLateDeliveries > 5;
    }
}
```

Refactored into:

```
// ...
{
    () {
        numberOfRowsLateDeliveries > 5 ? 2 : 1;
    }
}
```

Can be used to eliminate: **Speculative Generality**.

1. Composing Methods: Extract Variable

You have an expression that's hard to understand.

```
    () {
((platform.toUpperCase().indexOf("MAC") > -1) &&
 (browser.toUpperCase().indexOf("IE") > -1) &&
 wasInitialized() && resize > 0 )
{
    // do something
}
```

1. Composing Methods: Extract Variable

You have an expression that's hard to understand.

```
() {  
  ((platform.toUpperCase().indexOf("MAC") > -1) &&  
   (browser.toUpperCase().indexOf("IE") > -1) &&  
   wasInitialized() && resize > 0)  
  {  
    // do something  
  }  
}
```

Refactored into:

```
() {  
  isMacOs = platform.toUpperCase().indexOf("MAC") > -1;  
  isIE = browser.toUpperCase().indexOf("IE") > -1;  
  wasResized = resize > 0;  
  
  (isMacOs && isIE && wasInitialized() && wasResized) {  
    // do something  
  }  
}
```

Can be used to eliminate: Comment.

1. Composing Methods: Split Temporary Variable

```
temp = 2 * (height + width);
System.out.println(temp);

temp = height * width;
System.out.println(temp);
```

1. Composing Methods: Split Temporary Variable

```
temp = 2 * (height + width);
System.out.println(temp);

temp = height * width;
System.out.println(temp);
```

Refactored into:

```
perimeter = 2 * (height + width);
area = height * width;

System.out.println(perimeter);
System.out.println(area);
```

2. Moving Features between Objects (I)

Move **functionality** between classes, create new classes, and hide implementation details from public access.

These ones are about **moving** methods, fields and classes to their **correct** place:

- **Move Method** A method is used more in another class than in its own class.
- **Move Field** A field is used more in another class than in its own class.
- **Extract Class** When one class does the work of two, awkwardness results.
- **Inline Class** A class does almost nothing and is not responsible for anything, and no additional responsibilities are planned for it.

2. Moving Features between Objects (II)

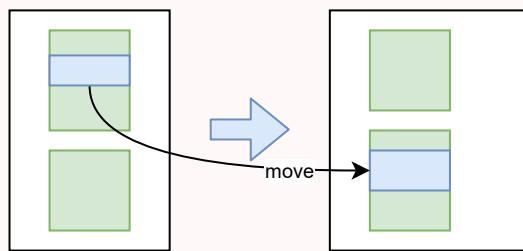
Move **functionality** between classes, create new classes, and hide implementation details from public access.

These ones are about **untangling** class associations by changing the way they are organized:

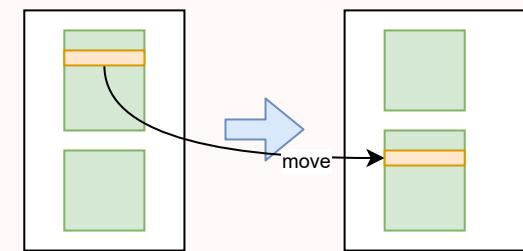
- **Hide Delegate** The client gets object B from a field or method of object A. Then the client calls a method of object B.
- **Remove Middle Man** A class has too many methods that simply delegate to other objects.
- **Introduce Foreign Method** A utility class does not contain the method that you need and you cannot add the method to the class.
- **Introduce Local Extension** Add the method to a client class and pass an object of the utility class to it as an argument.

2. Moving Features between Objects (III)

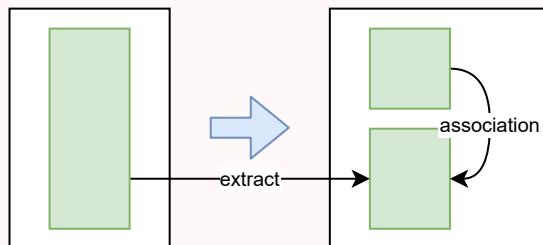
Move Method



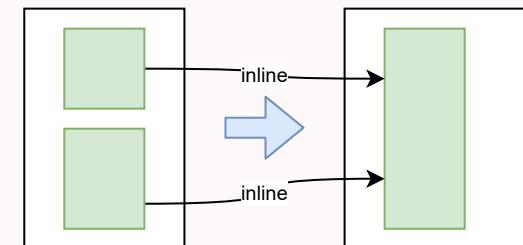
Move Field



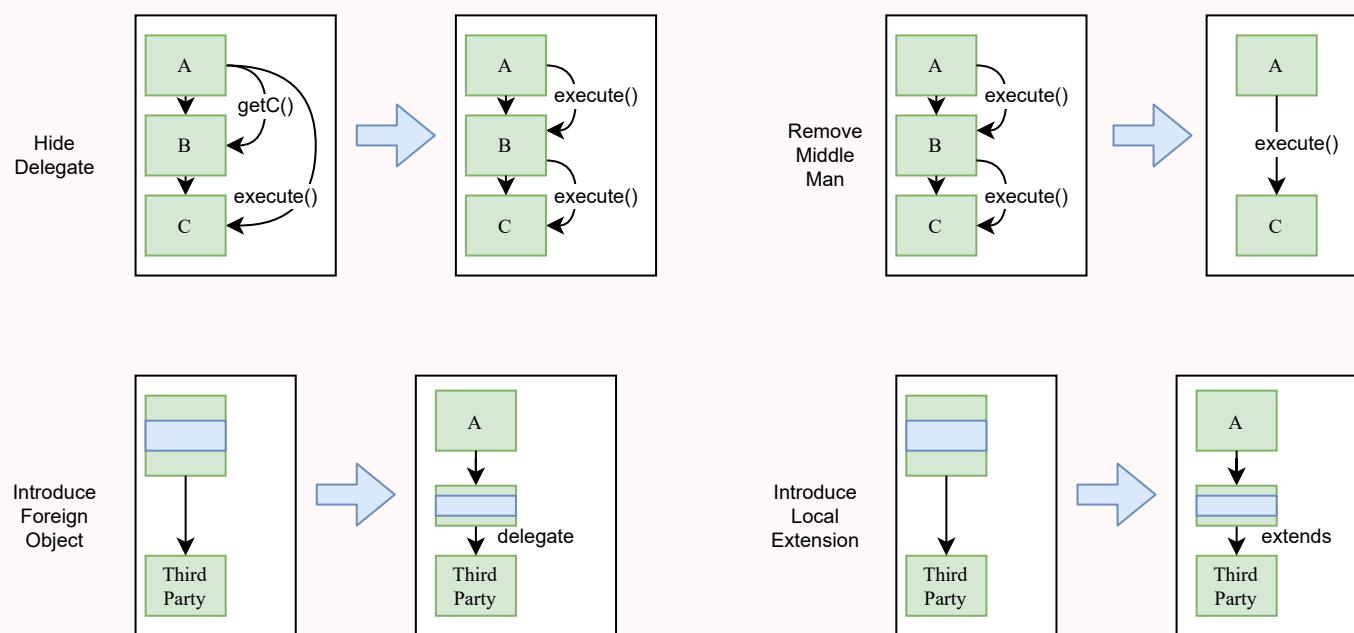
Extract Class



Inline Class



2. Moving Features between Objects (IV)



3. Organizing Data (I)

Helping with data handling by replacing primitives with rich class functionality.

- **Self Encapsulate Field** You use direct access to private fields inside a class.
- **Replace Data Value with Object** A class contains a data field that has its own behavior and associated data.
- **Change Value to Reference** You have many identical instances of a single class that you need to replace with a single object.
- **Change Reference to Value** You have a reference object that is too small and infrequently changed to justify managing its life cycle.

3. Organizing Data (II)

Helping with data handling by replacing primitives with rich class functionality.

- **Replace Array with Object** You have an array that contains various types of data.
- **Duplicate Observed Data** Is domain data stored in classes responsible for the GUI?
- **Change Unidirectional Association to Bidirectional** You have two classes that each need to use the features of the other, but the association between them is only unidirectional.
- **Change Bidirectional Association to Unidirectional** You have a bidirectional association between classes, but one of the classes does not use the other's features.

3. Organizing Data (III)

Helping with data handling by replacing primitives with rich class functionality.

- **Replace Magic Number with Symbolic Constant** Your code uses a number that has a certain meaning to it.
- **Encapsulate Field** You have a public field.
- **Encapsulate Collection** A class contains a collection field and a simple getter and setter for working with the collection.
- **Replace Type Code with Class** A class has a field that contains type code. The values of this type are not used in operator conditions and do not affect the behavior of the program.

3. Organizing Data (IV)

Helping with data handling by replacing primitives with rich class functionality.

- **Replace Type Code with Subclasses** You have a coded type that directly affects program behavior (values of this field trigger various code in conditionals).
- **Replace Type Code with State/Strategy** You have a coded type that affects behavior but you cannot use subclasses to get rid of it.
- **Replace Subclass with Fields** You have subclasses differing only in their (constant-returning) methods.

4. Simplifying Method Calls (I)

- **Rename Method** The name of a method does not explain what the method does.
- **Add Parameter** A method does not have enough data to perform certain actions.
- **Remove Parameter** A parameter is not used in the body of a method.
- **Separate Query from Modifier** Do you have a method that returns a value but also changes something inside an object?

4. Simplifying Method Calls (II)

- **Parameterize Method** Multiple methods perform similar actions that are different only in their internal values, numbers or operations.
- **Replace Parameter with Explicit Methods** A method is split into parts, each of which is run depending on the value of a parameter.
- **Preserve Whole Object** You get several values from an object and then pass them as parameters to a method.
- **Replace Parameter with Method Call** Before a method call, a second method is run and its result is sent back to the first method as an argument. But the parameter value could have been obtained inside the method being called.

4. Simplifying Method Calls (III)

- **Introduce Parameter Object** Your methods contain a repeating group of parameters.
- **Remove Setting Method** The value of a field should be set only when it is created, and not change at any time after that.
- **Hide Method** A method is not used by other classes or is used only inside its own class hierarchy.

4. Simplifying Method Calls (IV)

- **Replace Constructor with Factory Method** You have a complex constructor that does something more than just setting parameter values in object fields.
- **Replace Error Code with Exception** A method returns a special value that indicates an error?
- **Replace Exception with Test** You throw an exception in a place where a simple test would do the job?

5. Simplifying Conditional Expressions (I)

- **Decompose Conditional** You have a complex conditional (if-then/else or switch).
- **Consolidate Conditional Expression** You have multiple conditionals that lead to the same result or action.
- **Consolidate Duplicate Conditional Fragments** Identical code can be found in all branches of a conditional.
- **Remove Control Flag** You have a boolean variable that acts as a control flag for multiple boolean expressions.

5. Simplifying Conditional Expressions (II)

- **Replace Nested Conditional with Guard Clauses** You have a group of nested conditionals and it is hard to determine the normal flow of code execution.
- **Replace Conditional with Polymorphism** You have a conditional that performs various actions depending on object type or properties.
- **Introduce Null Object** Since some methods return null instead of real objects, you have many checks for null in your code.
- **Introduce Assertion** For a portion of code to work correctly, certain conditions or values must be true.

Decompose Conditional

```
(date.before(SUMMER_START) || date.after(SUMMER_END)) {  
    charge = quantity * winterRate + winterServiceCharge;  
}  
{  
    charge = quantity * summerRate;  
}
```

Decompose Conditional

```
(date.before(SUMMER_START) || date.after(SUMMER_END)) {  
    charge = quantity * winterRate + winterServiceCharge;  
}  
{  
    charge = quantity * summerRate;  
}
```

Refactored into:

```
(isSummer(date)) {  
    charge = summerCharge(quantity);  
}  
{  
    charge = winterCharge(quantity);  
}
```

Can be used to eliminate: **Long Method**.

Introduce Null Object

```
(customer == null) {  
    plan = BillingPlan.basic();  
}  
else {  
    plan = customer.getPlan();  
}
```

Introduce Null Object

```
(customer == null) {  
    plan = BillingPlan.basic();  
}  
else {  
    plan = customer.getPlan();  
}
```

Refactored into:

```
{  
    () {  
        ;  
    }  
    Plan () {  
        NullPlan();  
    }  
}  
  
customer = (order.customer != null) ?  
    order.customer : NullCustomer();  
  
plan = customer.getPlan();
```

Can be used to eliminate: **Switch Statements and Temporary Field.**

Replace Nested Conditional with Guard Clauses

```
        () {  
    result;  
    (isDead){  
        result = deadAmount();  
    }  
    {  
        (isSeparated){  
            result = separatedAmount();  
        }  
        {  
            (isRetired){  
                result = retiredAmount();  
            }  
            {  
                result = normalPayAmount();  
            }  
        }  
    }  
    result;  
}
```

```
        () {  
    (isDead){  
        deadAmount();  
    }  
    (isSeparated){  
        separatedAmount();  
    }  
    (isRetired){  
        retiredAmount();  
    }  
    normalPayAmount();  
}
```

Consolidate Duplicate Conditional Fragments

```
(isSpecialDeal()) {  
    total = price * 0.95;  
    send();  
}  
    {  
        total = price * 0.98;  
        send();  
}
```

Consolidate Duplicate Conditional Fragments

```
(isSpecialDeal()) {  
    total = price * 0.95;  
    send();  
}  
    {  
        total = price * 0.98;  
        send();  
    }
```

Refactored into:

```
(isSpecialDeal()) {  
    total = price * 0.95;  
}  
    {  
        total = price * 0.98;  
    }  
send();
```

Can be used to eliminate: **Duplicate Code**.

6. Dealing with Generalization (I)

- **Pull Up Field** Two classes have the same field.
- **Pull Up Method** Your subclasses have methods that perform similar work.
- **Pull Up Constructor Body** Your subclasses have constructors with code that is mostly identical.

6. Dealing with Generalization (II)

- **Push Down Method** Is behavior implemented in a superclass used by only one (or a few) subclasses?
- **Push Down Field** Is a field used only in a few subclasses?

6. Dealing with Generalization (III)

- **Extract Subclass** A class has features that are used only in certain cases.
- **Extract Superclass** You have two classes with common fields and methods.
- **Extract Interface** Multiple clients are using the same part of a class interface. Another case: part of the interface in two classes is the same.

6. Dealing with Generalization (IV)

- **Collapse Hierarchy** You have a class hierarchy in which a subclass is practically the same as its superclass.
- **Form Template Method** Your subclasses implement algorithms that contain similar steps in the same order. **Replace Inheritance with Delegation** You have a subclass that uses only a portion of the methods of its superclass (or it's not possible to inherit superclass data).
- **Replace Delegation with Inheritance** A class contains many simple methods that delegate to all methods of another class.

Java Generics

André Restivo

Index

Introduction

Type Variables

Generic Methods

Generic Classes

Variance

Wildcards

Observer Example

Transformation Example

Introduction

Java without Generics

Without generics, we would write code like this:

```
List wordList = ArrayList();
wordList.add("ABCD");
String word = wordList.get(0);
```

Only to find the compiler complaining about that last line.

This happens because, unqualified Lists, store **Objects**. So, `wordList.get(0)` returns an **Object**.

This means we need to cast it to a String first:

```
String word = (String)wordList.get(0);
```

Why we need Generics

Without generics, the following code will throw a `ClassCastException`:

```
List wordList =     ArrayList();
wordList.add(123);
String word = (String) wordList.get(0);
```

It would be much nicer if we could detect this kind of bugs in **compile-time**.

What generics provide is **Type-safety**:

```
List<String> wordList =     ArrayList<>();
wordList.add(123); // add (String) in List cannot be applied to (int)
String word = wordList.get(0);
```

Now, besides **not** needing the explicit `cast`, trying to add an integer to the list, results in a **compile-time** error.

Type Variables

Type Variables

- A **type variable** is an unqualified identifier. Unlike String, which is qualified, an unqualified identifier doesn't tell explicitly which class we are talking about.
- A **class, interface, method or constructor** is generic if it declares one or more type variables.

Type variables are declared using the **diamond operator**:

```
<T>
```

Type Variable Names

There are some naming conventions usually associated with the use of generics.

Normally, type variable names are single, uppercase letters to make it easily distinguishable from java variables:

- E – **Element** (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
- K – **Key** (Used in Map)
- N – **Number**
- T – **Type**
- V – **Value** (Used in Map)
- S,U,V etc. – 2nd, 3rd, 4th types

Generic Methods

Methods without Generics

Consider that we want to create a function that adds an **Array of Strings** to a **List of Strings**:

```
(String[] a, List<String> l) {  
    String s : a) l.add(s);  
}
```

Now imagine we want to extend this method to **all types** of objects.

Methods without Generics

Without generics, we would do it like this:

```
        (Object[] a, List l) {  
    (Object o : a) l.add(o);  
}
```

Two problems:

- We can't guarantee in **compile-time** the type of the **Array**.
- There are no guarantees about the types inside the **List** we are getting back.

So if we want to retrieve the values from the **List**, we have to cast them to **String** and risk a **ClassCastException**.

```
String[] a = {"ABC", "DEF"}; List l =     ArrayList();  
  
arrayToList(a, l);  
  
    (Object s : l)  
System.out.println((String)s);
```

Methods without Generics

And even if we swear to always use parameterized Lists:

```
String[] a = {"ABC", "DEF"}; List<String> l =     ArrayList<>();  
arrayToList(a, l);  
  
    (String s : l)  
    System.out.println(s);
```

Something like this would still be possible:

```
String[] a = {"ABC", "DEF"}; List<Integer> l =     ArrayList<>();  
arrayToList(a, l);  
  
    (Integer i : l)  
    System.out.println(i);
```

And we would get a **ClassCastException** again!

Generic Methods

Using generics, we can solve this problem by creating a parameterized function:

```
<T>          (T[] a, List<T> l) {  
    (T o : a) l.add(o);  
}
```

Notice how we declared that the function is **parameterized** by a **type variable** called **T**. Now we can call this method with any kind of **List** whose element type is a **supertype** of the element type of the **Array**.

```
String[] a = {"ABC", "DEF"}; List<String> l = ArrayList<>();  
  
arrayToList(a, l);  
  
(String s : l)  
System.out.println(s);
```

And if we try to use **incompatible** types for the **Array** and **List**, we will get a **compile-time** error.

Generic Classes

Generic Classes

We can define our **own classes** with generics type.

A generic type is a **class or interface** that is **parameterized over types**:

```
< > {  
T something;  
  
    (T something) {  
        .something = something;  
    }  
  
    T () {  
        .something;  
    }  
}
```

Using it:

```
Box<String> box =      Box<>();  
box.put("ABC");          // Only works with strings  
String something = box.get(); // Only returns strings
```

Extending Generic Types

We can also create classes that extend generic types:

```
< > {  
String name;  
        (String name) {  
    .name = name;  
}  
    String () {  
        .name;  
}  
}
```

We can even create non-parameterized types based on parameterized types:

```
< > { }
```

Variance

Or a closer look at subtyping

Variance

Variance refers to how **subtyping** between more **complex types** relates to subtyping between their components:

- **Covariance**: if it preserves the ordering of types (accepts subtypes).
- **Contravariance**: if it reverses the ordering of types (accepts supertypes).
- **Invariance**: if it only accepts the specific type.

Basic Types

Basic Java types are **covariant**. So this is possible:

```
{ }  
{ }  
  
Zebra zebra = Zebra();  
Animal animal = zebra;
```

All Zebras are Animals. But this is not:

```
{ }  
{ }  
{ }  
  
Animal animal = Giraffe(); // Ok.  
Zebra zebra = animal; // Incompatible types.
```

Not all Animals are Zebras...

Arrays

Arrays are also covariant. So this is possible:

```
{ }  
{ }  
  
Zebra[] zebras = Zebra[10];  
Animal[] animals = zebras;
```

All groups of Zebras are a group of Animals. But this is not:

```
{ }  
{ }  
{ }  
  
Animal[] animals = Giraffe[10]; // Ok.  
Zebra[] zebras = animals; // Incompatible types.
```

Not all groups of Animals are a group of Zebras.

Arrays

But we need to be careful:

```
{ }  
{ }  
{ }  
  
Zebra[] zebras = Zebra[10];  
Animal[] animals = zebras;  
  
animals[0] = Zebra(); // Ok  
animals[1] = Giraffe(); // ArrayStoreException
```

This will throw an **exception** at **runtime**, as the **animals** array is still just an array of zebras.

Generics

Due to something called "**type erasure**", Java has no way of knowing at runtime the type information of the type parameters.

For historical reasons, **arrays are covariant** but we have to deal with runtime exceptions.

For this reason, Java **generics are invariant**. We can deal with this problem in a more sophisticated way using **wildcards**.

```
List<Zebra> zebras = ArrayList<>();
List<Zebra> theSameZebras = zebras; // Ok.
List<Animal> animals = zebras; // Incompatible types (compile-time error).
```

Wildcards

Wildcards

Instead of a **type variable** like `<T>`, we can use a wildcard like `<?>` to represent an unknown type:

```
List<Zebra> zebras = ArrayList<>();
List<Zebra> theSameZebras = zebras; // Ok.
List<?> animals = zebras;
```

Or create a method that receives a list of "unknowns".

```
(List<?> someList) {
    (Object o : someList)
        System.out.println(o);
}
```

We can use it like this:

```
List<Zebra> zebras = ArrayList<>();
printList(zebras);
```

Wildcards

However, this does not work:

```
List<?> zebras =      ArrayList<>();  
zebras.add(      Zebra()); // add (<?>) in List cannot be applied
```

A list of "unknowns" is still a list of something.

We just don't know what it is.

We can't just start stuffing zebras inside one.

Bounded Wildcards

We can, however, use extends and super to create more flexible lists.

A list of animals or any subtype (upper-bounded):

```
List<? extends Animal> animals = ArrayList<>();
```

A list of animals or any supertype (lower-bounded):

```
List<? super Animal> animals = ArrayList<>();
```

Upper-bounded wildcards are **covariant**, while lower-bounded wildcards are **contravariant**.

Read and Write

Covariant types are "read-only" (kind of):

```
List<? extends Animal> animals =      ArrayList<>();
(Animal a : animals)
System.out.println(a);
animals.add(    Zebra()); // add (<? extends Animal>) in List
                  // cannot be applied to (Zebra)
((List<Zebra>)animals).add(    Zebra()); // Ok.
                                         // But we can get a ClassCastException
```

We cannot be certain that the list is of **Zebras**. This is not a list of **Animals**, it is a list of some "unknown" Animal type.

Contravariant types are "write-only" (kind of):

```
List<? extends Animal> animals =      ArrayList<>();
animals.add(    Zebra());
(Animal a : animals)
System.out.println(a); // Incompatible types.
(Object a : animals)
System.out.println(a); // Ok. But we don't know the real type.
```

Examples

If we don't care about the data inside a collection:

```
(Collection<?> collection) {  
    collection.size();  
}
```

If we just need a list where we can put animals. Even if it is a List<Object>.

```
(Animal animal, List<? extends Animal> list) {  
    list.add(animal);  
}
```

If we want to make sure the List contains animals. It can be a List<Zebra>.

```
(List<? extends Animal> list) {  
    (Animal animal : list)  
        animal.eat();  
}
```

Observer Example

Generic Observer Interface

```
< > {  
    (T observable);  
}
```

Generic Observable Class

```
        < > {
List<Observer<T>> observers;

    () {
.observers =      ArrayList<>();
}

    (Observer<T> observer) {
observers.add(observer);
}

    (Observer<T> observer) {
observers.remove(observer);
}

    (T subject) {
(Observer<T> observer : observers)
observer.changed(subject);
}
}
```

Light

```
        <     > {
    turnedOn;

    (      turnedOn) {
.turnedOn = turnedOn;
}

() {
    turnedOn;
}

(      turnedOn) {
.turnedOn = turnedOn;
.notifyObservers(  );
}
}
```

Client

```
        {
            (String[] args) {
Light light =      Light(    );
light.addObserver(      Observer<Light>() {
    @Override
        (Light light) {
    System.out.println("LIght: " + light.isTurnedOn());
}
});  
light.setTurnedOn(    );
light.setTurnedOn(    );
light.setTurnedOn(    );
    }
}
```

Transformation Example

Generic Transformation Interface

Transforms one variable into another having the same type.

```
< > {  
T      (T argument);  
}
```

Generic Transformation Line

Does a series of transformations in sequence:

```
        < > {
List<Transformation<T>> transformations;

        () {
.transformations =      ArrayList<>();
}

        (Transformation<T> transformation) {
.transformations.add(transformation);
}

        T      (T value) {
T current = value;
        (Transformation<T> transformation : transformations)
current = transformation.execute(current);
        current;
}
}
```

Client

```
TransformationLine<Integer> tl = TransformationLine<>();  
  
tl.addTransformation( Transformation<Integer>() {  
    @Override  
    Integer (Integer argument) {  
        argument + 1;  
    }  
});  
  
tl.addTransformation( Transformation<Integer>() {  
    @Override  
    Integer (Integer argument) {  
        argument * 2;  
    }  
});  
  
tl.addTransformation( Transformation<Integer>() {  
    @Override  
    Integer (Integer argument) {  
        argument + 5;  
    }  
});  
  
System.out.println(tl.execute(10));
```

UML

Sequence Diagrams

André Restivo

Index

Introduction

Lifelines

Messages

Fragments

Gates

Introduction

Types of Diagrams

In UML, there are two basic categories of diagrams:

- **Structure** diagrams show the static structure of the system being modeled: *class*, *component*, *deployment*, *object* diagrams, ...
- **Behavioral** diagrams show the dynamic behavior between the objects in the system: *activity*, *use case*, *communication*, *state machine*, *sequence* diagrams, ...

Sequence Diagrams

Sequence diagrams depict the interaction between **objects** in a **sequential order**.

The main focus of sequence diagrams is the exchange of **messages** between objects and their **lifelines**.

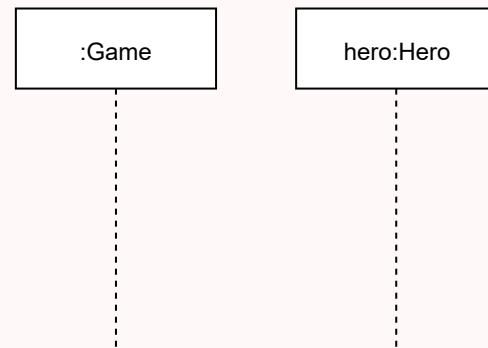
Sequence diagrams are used **either** to model generic interactions (showing **all possible paths** through the interaction) or specific instances of a interaction (showing **just one path** through the interaction).

Lifelines

Lifeline

Lifeline is a **named element** which represents an **individual participant** in the interaction.

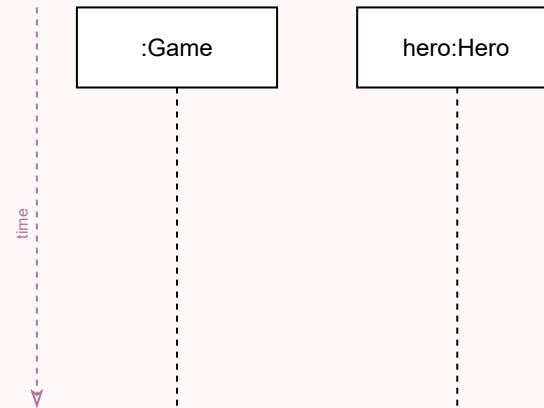
A lifeline is composed by an **head**, a rectangle that identifies the participant element, and a vertical dashed **line**.



The element can be an **anonymous** representative of a certain class, or a **named** one.

Axis

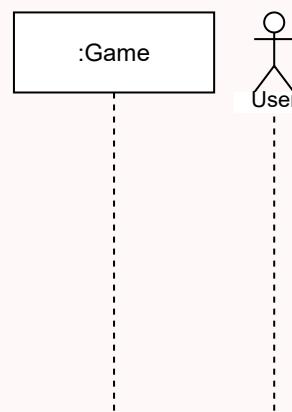
The **horizontal axis** of a sequence diagram represents the **object instances** (left to right) that participate in the interaction. Normally objects appear in the same order as they interact for the first time.



The **vertical axis** represents **time** (top to bottom). Time in a sequence diagram is all about **ordering, not duration**. The vertical space in an interaction diagram is not relevant for the duration of the interaction.

Actors

An Actor is always something (a system or person) that is **outside the scope** of the system.

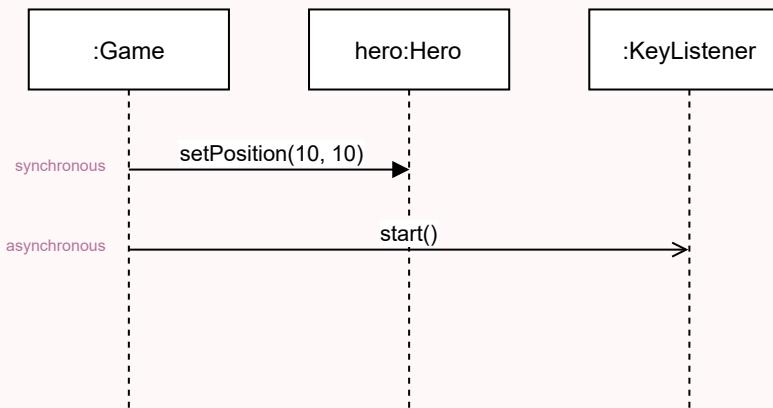


Actors are drawn as **stickman** figures (although they may not be users), and can be participants in sequence diagrams.

Messages

Messages

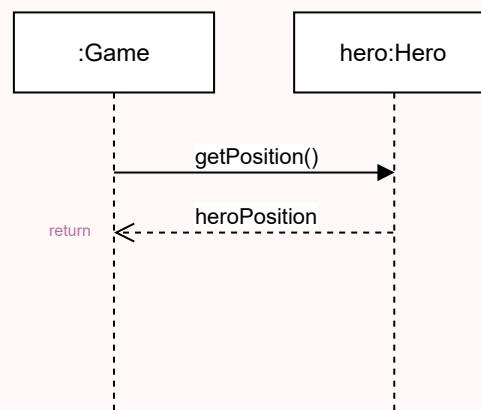
Messages are represented by a **line** from the **sending object's** lifeline to the **receiving object's** lifeline with a **solid arrowhead** (if a *synchronous* signal) or with a **stick arrowhead** (if an *asynchronous* signal).



The message/method name is placed **above** the arrowed line and represents an **operation/method** that the receiving object's class implements.

Return

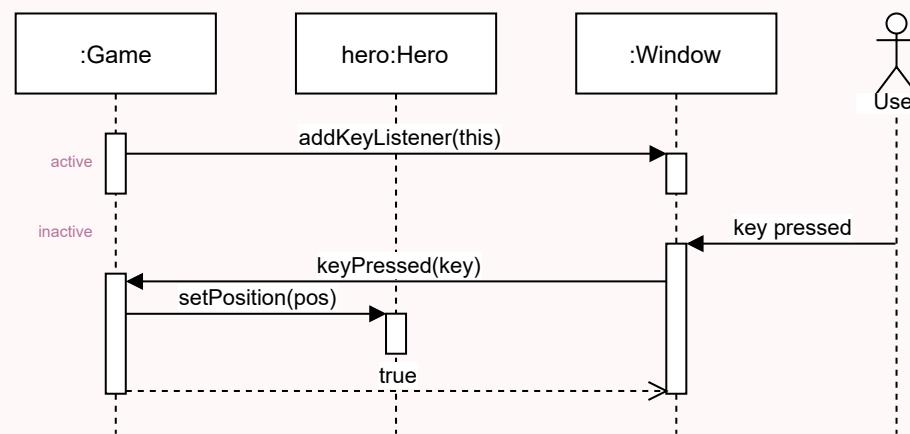
Return messages are **optional** and are represented by a **dashed line with a stick arrowhead**.



The return value, if needed, is place **above** the arrowed line. The returned value can either be a **concrete value** or just a **name**.

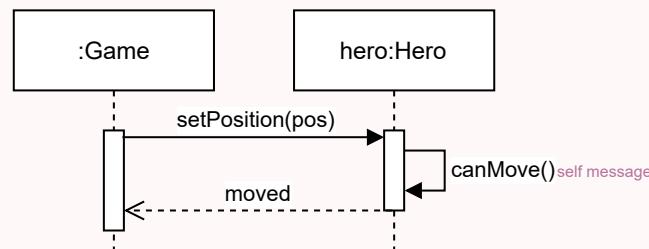
Activation

An *optional* thin rectangle on a lifeline represents the period during which an element is performing an operation.

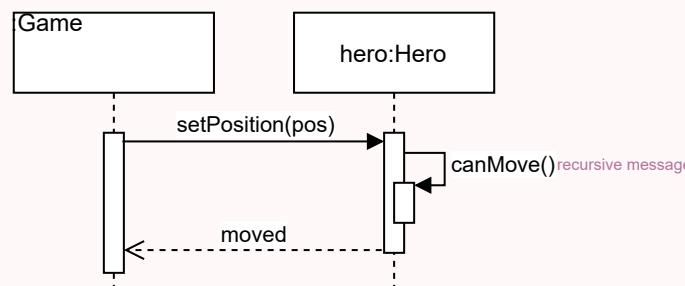


Self Message

An object can send a message to **itself** (a **self message**).

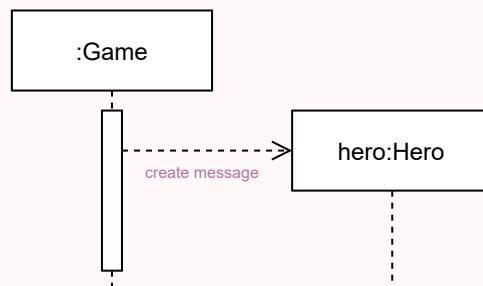


Optionally, you can represent the **recursive activation** created by this call. This can be useful if you want to show **which function** is interacting with other objects.



Create Message

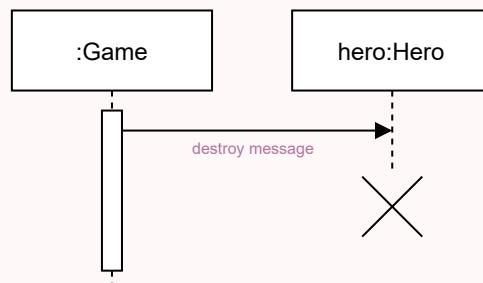
A **create message** is a kind of message that represents the instantiation of a lifeline.



They are represented with a **dashed line** with **stick arrowhead**.

Destroy Message

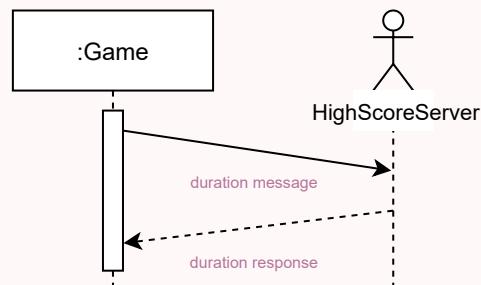
A **destroy message** is a kind of message that represents the destruction of a lifeline.



They don't have a specific representation besides the lifeline **terminating with a cross**.

Duration Message

Duration messages are used to indicate that a particular message should **not** be considered as instantaneous.

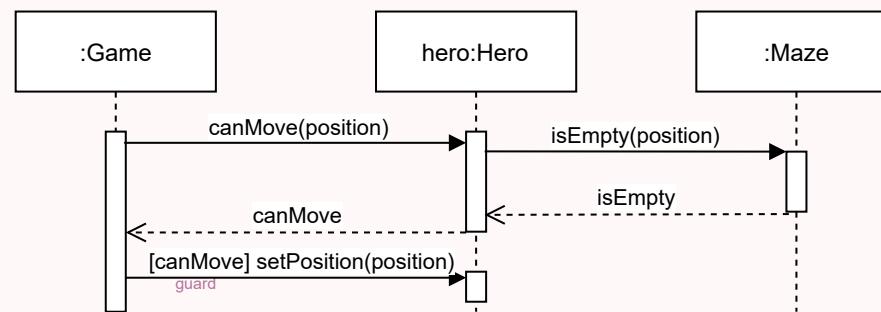


They are represented as a slanted line.

Guards

Sometimes we want to represent more **complex** interaction flows.

A **guard** is a **condition** that can be attached to a message. The message will be sent **only if** the condition is met.



Guards are written inside **square brackets**.

Combined fragments

Combined Fragments

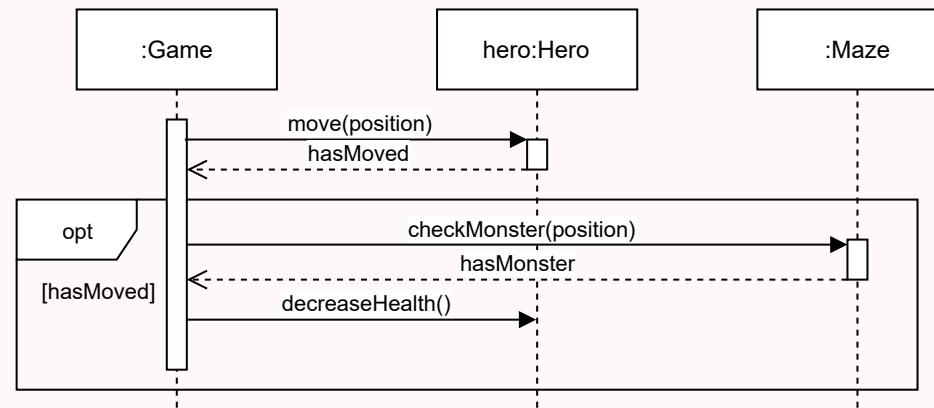
Sometimes **guards** are **not enough** to express the flow of a more **complex** sequence diagram.

A **combined** fragment is used to **group** sets of **messages** together to show **conditional** flow in a sequence diagram.

There are many types of interaction types for combined fragments. We will approach only the more useful.

Option Combination

Option combinations are used to designate a set of messages that will only be sent if a certain condition is met.

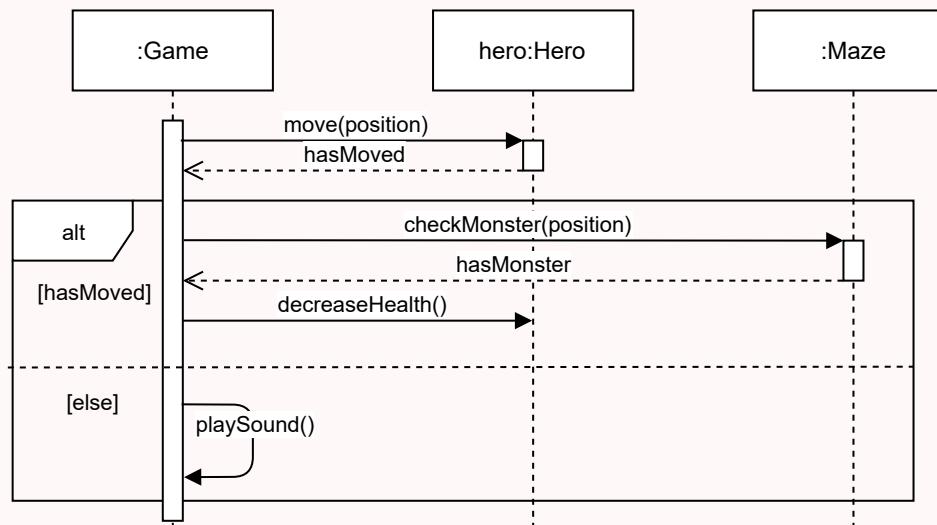


An alternative combination fragment element is drawn using a **frame** with the name "opt" (using guard like syntax)

Alternative Combination

Alternative combinations are used to designate a **mutually exclusive** choice between two or more message sequences.

An alternative combination fragment element is drawn using a **frame** with the name "alt".

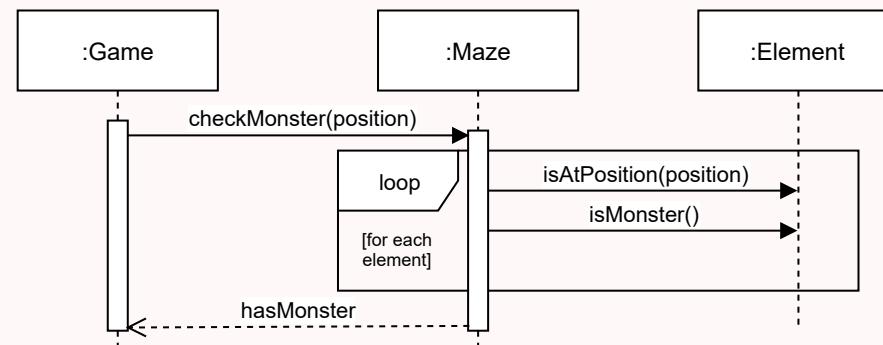


The frame is divided into **rectangles** representing alternative flows (using guard like syntax).

Loop Combination

Loop combinations are used to designate a set of messages that are to be sent a number of times.

An alternative combination fragment element is drawn using a **frame** with the name "loop".

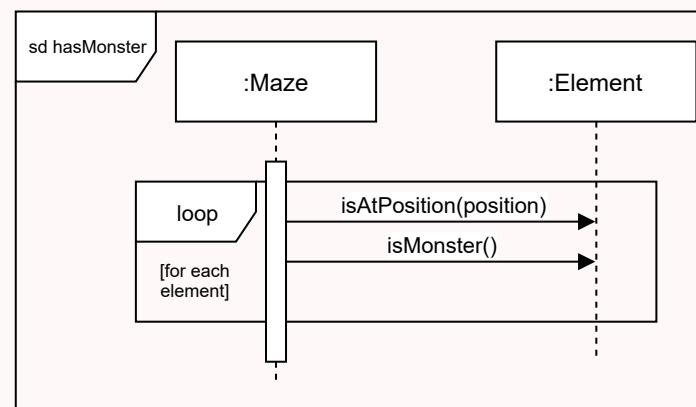


The number of iterations is defined inside square brackets (e.g. 5 times, for all elements, ...).

Gates

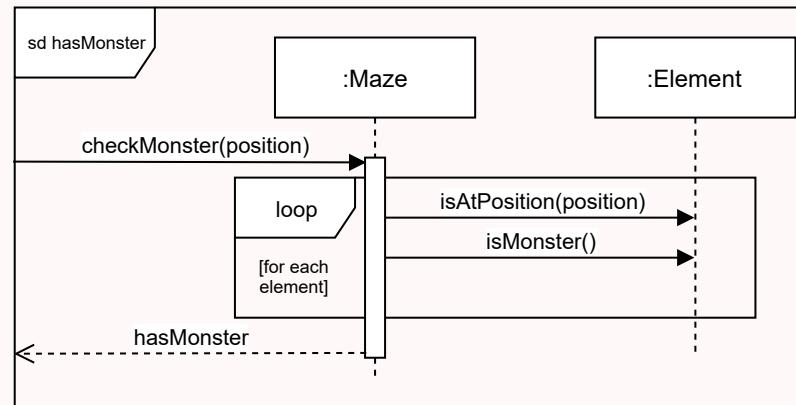
Frames

Sequence diagrams can be drawn inside frames so that we can give them a name (and something more...).



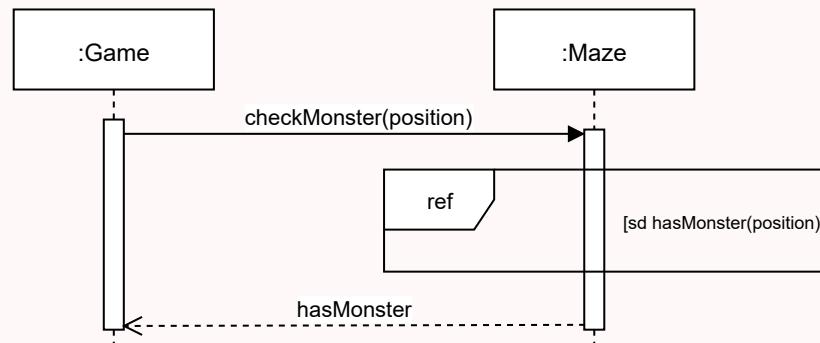
Gates

A **gate** is a message with one end connected to the sequence diagram's frame's edge and the other end connected to a **lifeline**.



References

Gates allow us to **reference** other sequence diagrams to create more **complex** ones.



The referenced diagram, receives the same parameters as its gates.

UML

Communication Diagrams

André Restivo

Index

Introduction

Objects

Messages

Sequence Expressions

Introduction

Types of Diagrams

In UML, there are two basic categories of diagrams:

- **Structure** diagrams show the static structure of the system being modeled: *class*, *component*, *deployment*, *object* diagrams, ...
- **Behavioral** diagrams show the dynamic behavior between the objects in the system: *activity*, *use case*, *communication*, *state machine*, *sequence* diagrams, ...

Communication Diagrams

Communication diagrams are a simplified version of sequence diagrams.

The main difference is that sequence diagrams are good at showing sequential logic but not that good at giving you a big picture view.

Objects

Objects

Objects are **named elements** which represent **individual participants** in the interaction.

An object is represented by a rectangle that identifies the participant element.



The element can be an **anonymous** representative of a certain class, or a **named** one.

Actors

An **Actor** is always something (a system or person) that is **outside the scope** of the system.

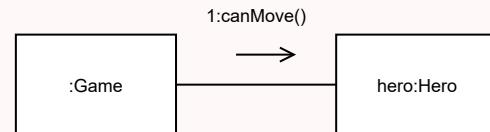


Actors are drawn as **stickman** figures (although they may not be users), and can be participants in communication diagrams.

Messages

Messages

Messages are represented by a **line** with an arrow above that indicates the direction of the message (and a sequence expression).



Sequence Expressions

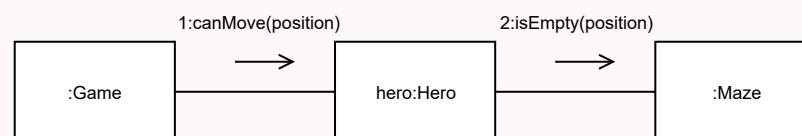
Sequence Expressions

The sequence expression is a dot (".") separated **list of sequence terms** followed by a colon (":") and **message name** after that:

```
term1.term2.term3:message
```

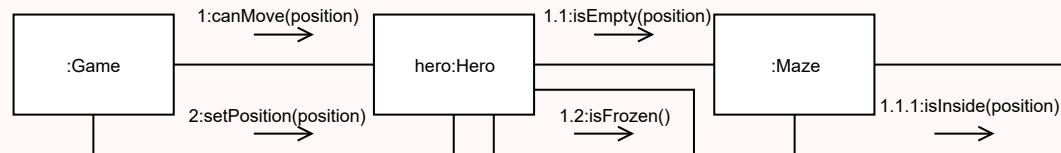
Each sequence term represents a **level** of procedural **nesting** within the **overall interaction**. Each sequence-term has the following syntax:

```
integer [ name ] [ recurrence ]
```



Sequence Order

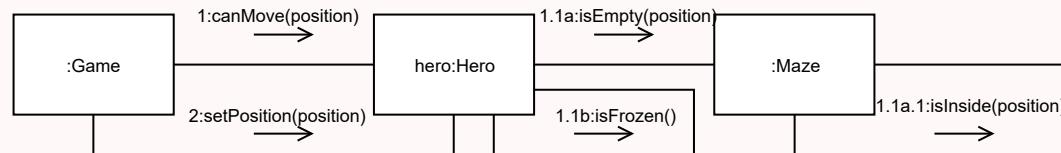
The integer represents the sequential order of the message within the next higher level of procedural calling (**activation**).



Messages that differ in one integer term are **sequential** at that level of nesting.

Sequence Name

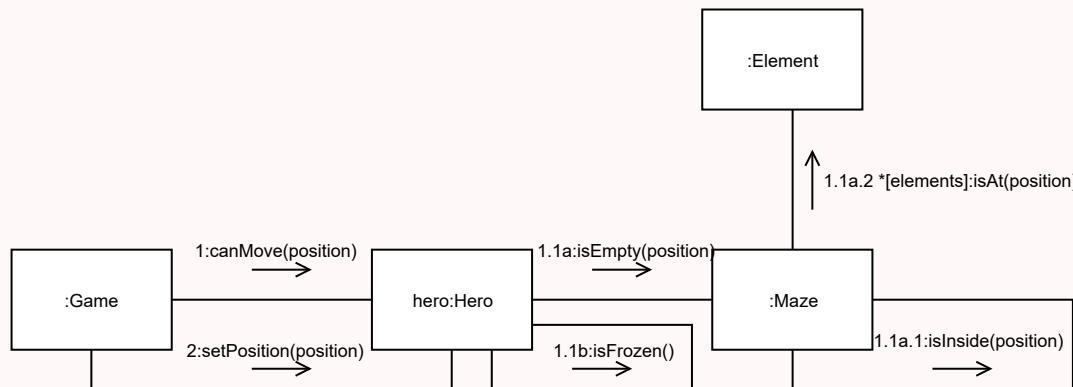
The **name** represents a **concurrent thread** of control. Messages that differ in the final name are **concurrent** at that level of nesting.



The `hero` instance send **both** requests (`1.1a` and `1.1b`) to the `Maze` object **concurrently**.

Sequence Recurrence

The recurrence of a sequence term can be a guard (a condition inside **square brackets**) or a loop (an **asterisk** followed by a condition inside **square brackets**).



Inheritance vs. Composition

André Restivo

Index

Motivation

Multiple Inheritance

Interfaces

Combination Classes

Java Defaults

Composition

Delegation

Motivation

Motivation

We want to create a system where we have several different types of **fruits**.

Some fruits are **edible**, some are **peelable** and some are **sliceable**.

Examples:

- An **orange** is *edible* and *peelable* but not *sliceable*.
- An **apple** is *edible*, *peelable* and *sliceable*.
- A **watermelon** is *edible* and *sliceable* but not *peelable*.
- A **poisonberry** (a real thing) is neither *edible*, *sliceable* or *peelable*.

Motivation

All fruits have a **weight** and a **color**.

And we want something like this to be possible:

```
public static void main
    new

private static void peelAndEat
    while
```

First Approach

We can start by imagining that we need classes **similar** to this:

| Fruit |
|---------------------|
| weight: double |
| color: String |
| getWeight(): double |
| getColor(): String |

| Sliceable |
|------------------|
| slices: int = 1 |
| getSlices(): int |
| slice(slices) |

| Peelable |
|---------------------|
| peeled: boolean |
| isPeeled(): boolean |
| peel() |

| Edible |
|-------------------------------|
| percentageEaten: double |
| getPercentageEaten() : double |
| eat(percentage: double) |

Code (Fruit)

```
public class Fruit
    private final double
    private final

        double
        this
        this

    public double getWeight
        return

    public      getColor
        return
```

Code (Edible)

```
public class Edible
    private double

    public Edible
        this

    public void eat double

    public double getPercentageEaten
        return
```

Code (Peelable)

```
public class Peelable
    private boolean

    public Peelable
        this      false

    public boolean isPeeled
        return

    public void peel
        this      true
```

Code (Sliceable)

```
public class Sliceable
    private int

    public Sliceable
        this

    public void slice int
        this

    public int getPieces
        return
```

Multiple Inheritance

Multiple Inheritance

One possible approach would be to use **multiple inheritance**.

Unfortunately multiple inheritance is **not supported** in many languages (including Java).

The argument behind this *controversial* decision, is that multiple inheritance adds **complexity** and suffers from **ambiguity** problems (namely the famous **diamond problem**).

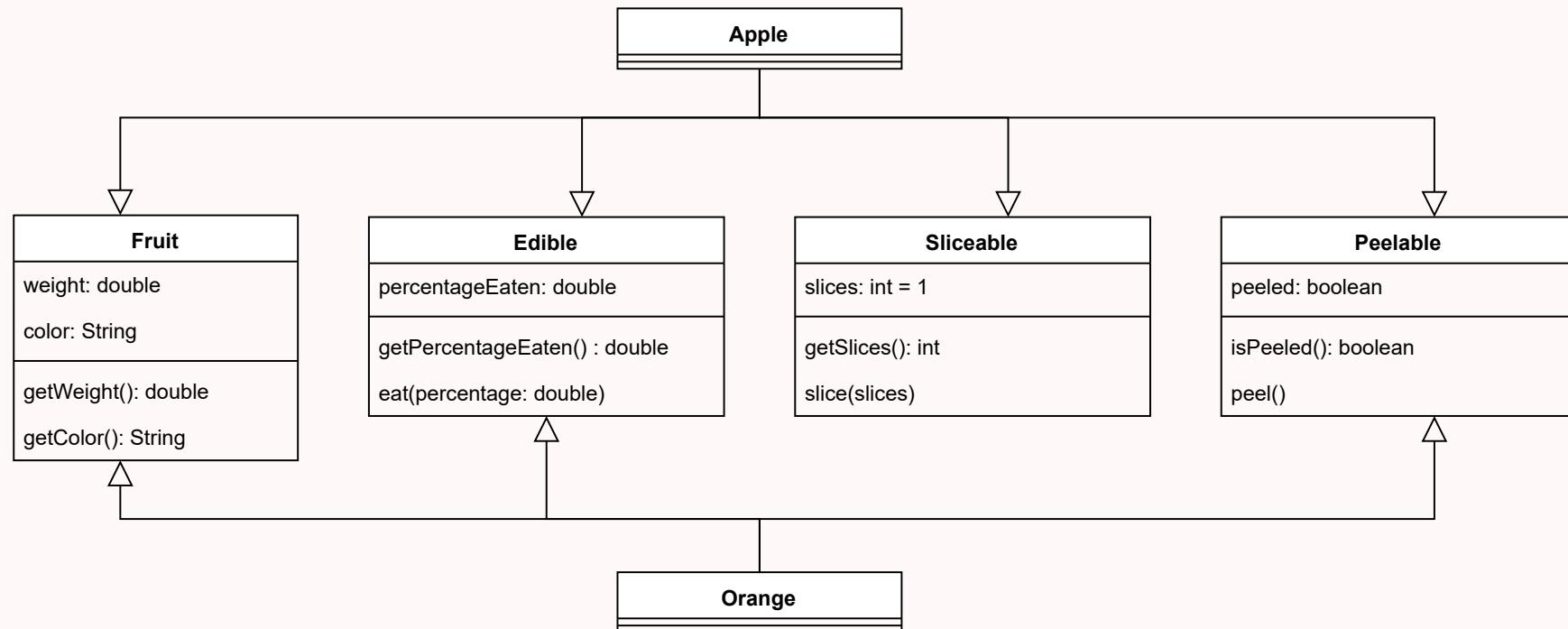
Disambiguation

Languages that allow *multiple inheritance* must have a **mechanism to disambiguate** which method should be called if two *super-classes* declare a method with the same signature.

This can be done in different ways:

- Not allowing multiple inheritance.
- Following the order they are declared.
- Explicitly by the developer.

Using Multiple Inheritance



Problems

Even if we could use *multiple inheritance*, this would not be possible:

```
public static void main
    new

private static void peelAndEat

    while
```

As not all fruits are **peelable** and **edible**.

Interfaces

Interfaces

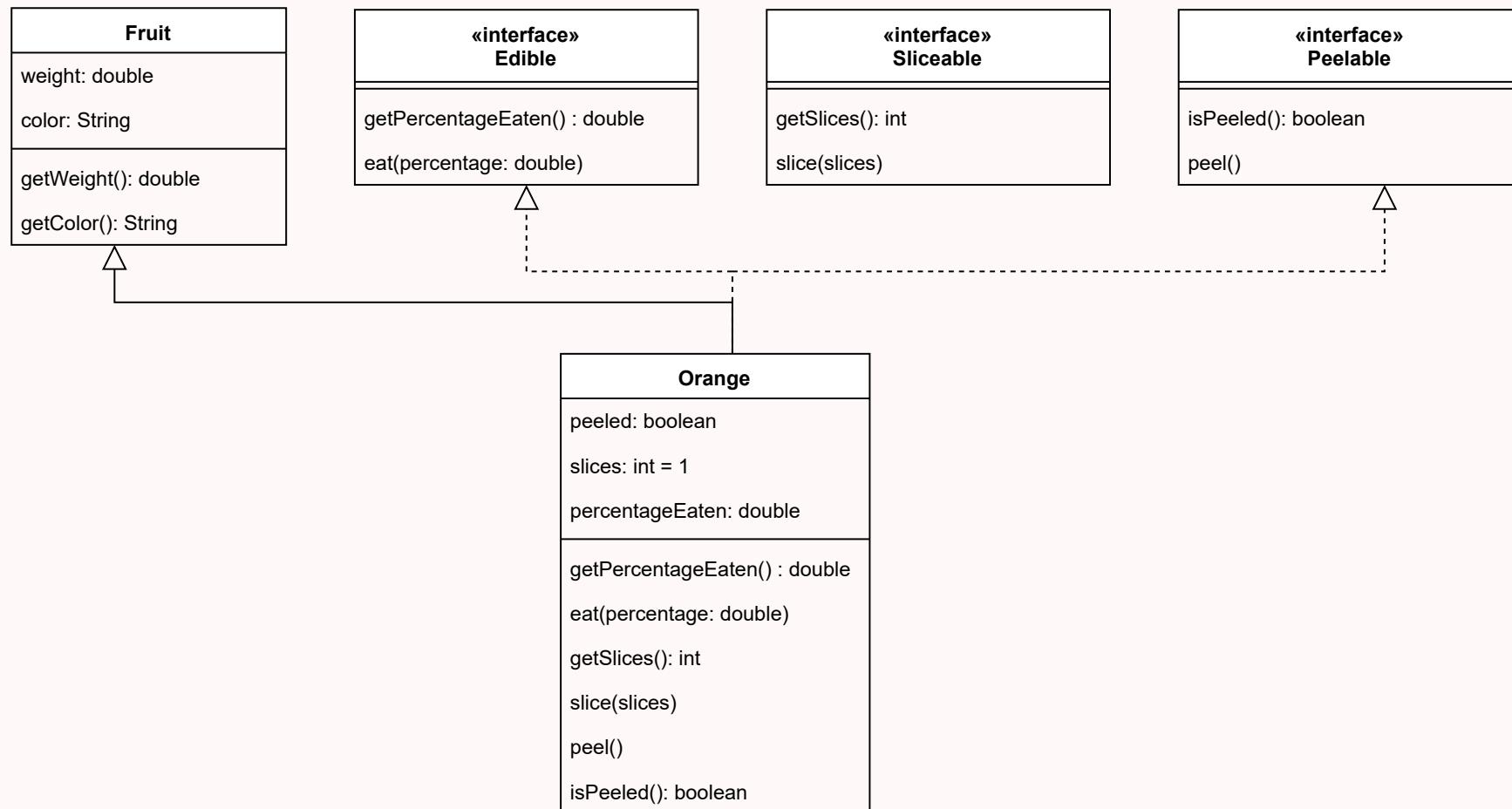
The alternative to *multiple inheritance* is to use **interfaces**.

Interfaces are structures that force classes to **implement** certain methods.

Methods in interfaces **don't have bodies**.

In Java, all variables declared inside interfaces are implicitly **public, static, and final**.

Using Interfaces



Problems

As **each class** must provide their own **implementation** of methods declared by the interfaces they implement, we end up with lots of **duplicate code**.

And we still **can't** do this:

```
public static void main
    new

private static void peelAndEat

    while
```

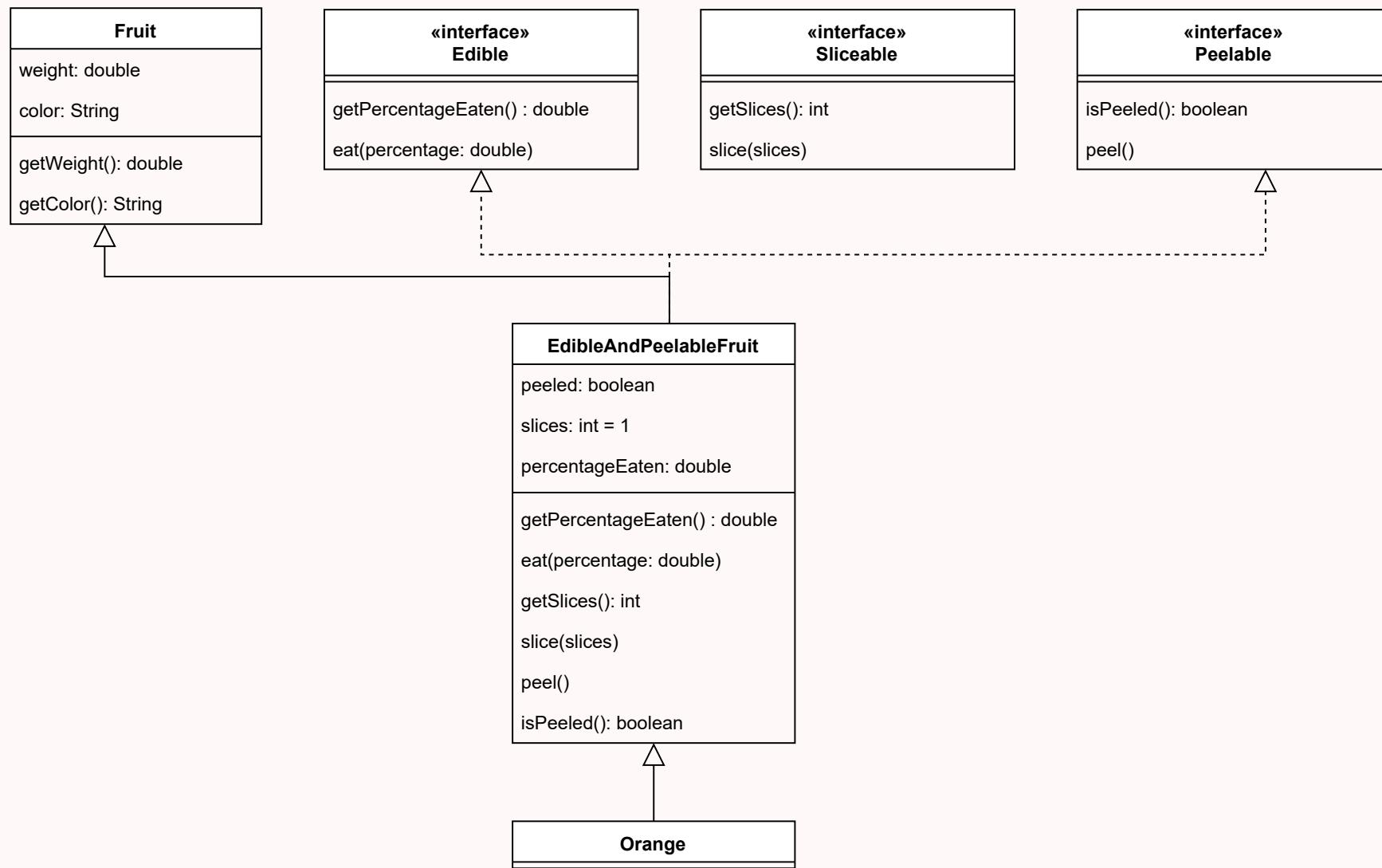
Combination Classes

Combination Classes

One solution would be to have abstract classes for all (or only those that we need) combinations of the *edible*, *peelable* and *sliceable* interfaces:

- EdibleFruit, PeelableFruit and SliceableFruit.
- EdibleAndPeelableFruit, EdibleAndSliceableFruit and PeelableAndSliceableFruit.
- EdiblePeelableAndSliceableFruit.

Using Combination Classes



Problems

As the number of interfaces that we want to implement grows, this quickly becomes impractical.

But at least we can do this:

```
public static void main
    new

private static void peelAndEat
    while
```

Defaults

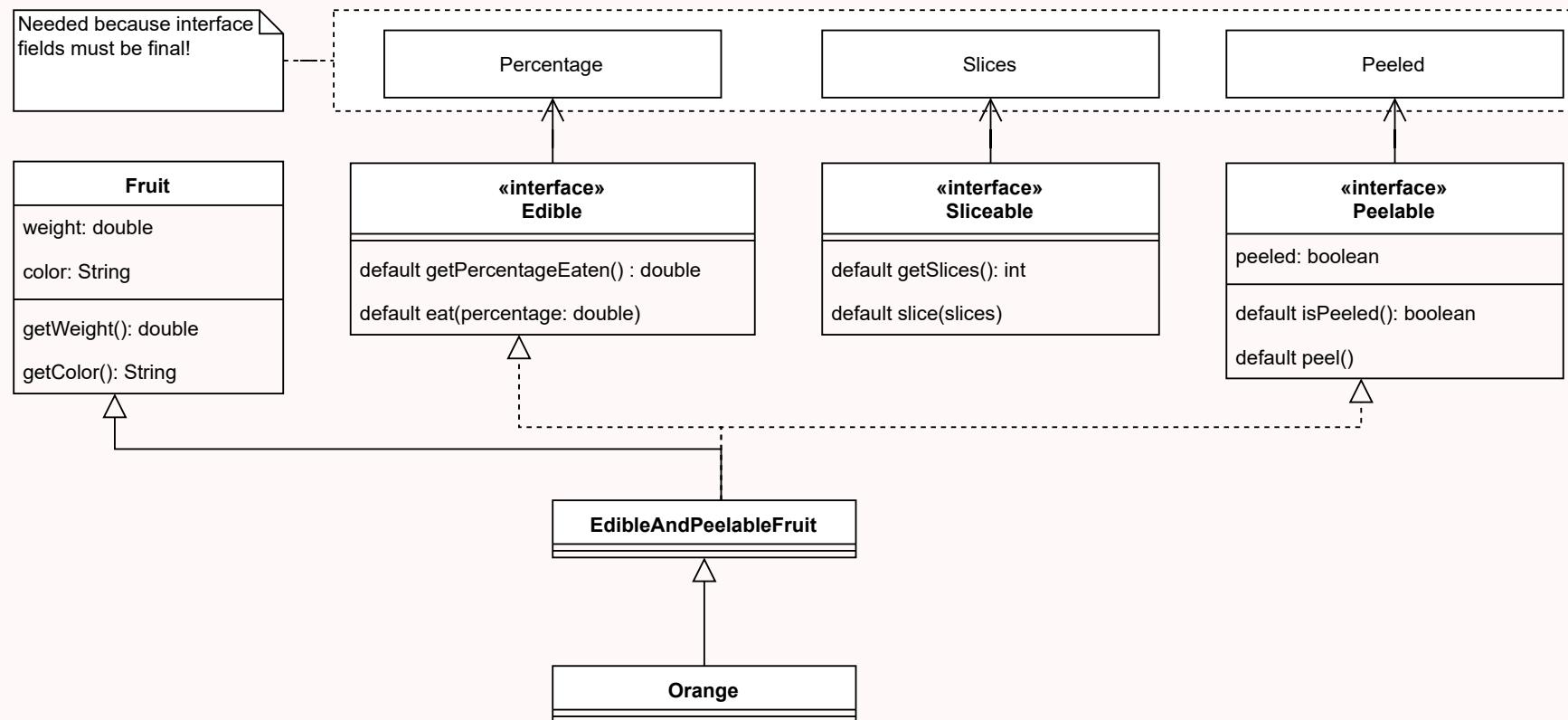
Defaults

Since Java 8, interfaces can have **default** implementations.

This means that interfaces **can** now declare method **bodies** that will be inherited by any classes implementing them.

But attributes still need to be **public**, **static**, and **final** (and this kind of defeats the purpose of having code in interfaces).

Using Defaults



Problems

We can **get around** the problem of all attributes being **final** by using **wrapper** classes.

But the only way to solve the **static** problem would be to have *maps* saving the data for each different instance. Which is overkill...

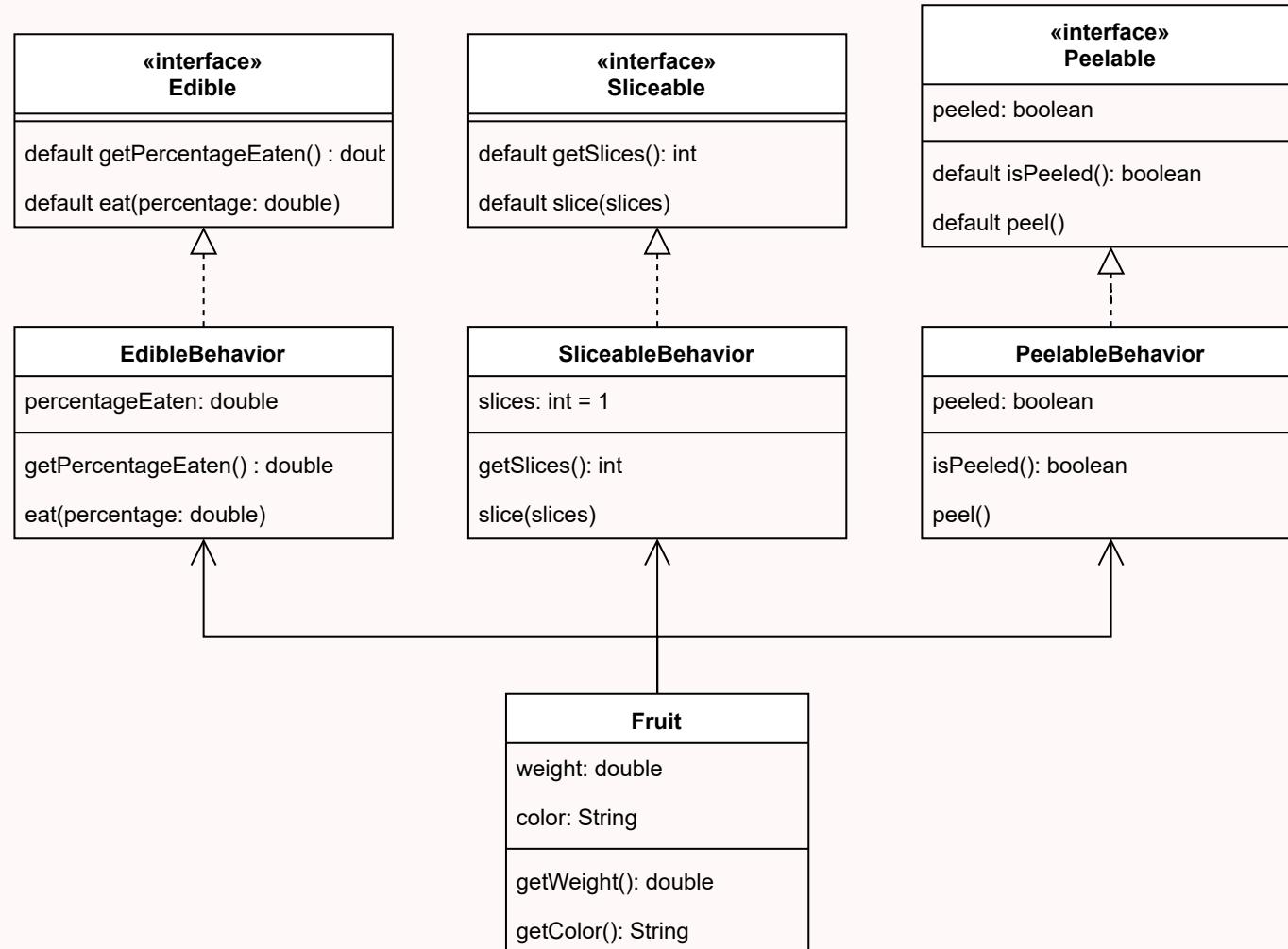
Composition

Composition

Composition over inheritance is the principle that classes should achieve **polymorphic** behavior and **code reuse** by their **composition** rather than by inheritance.

Instead of fruits inheriting from the *Edible*, *Peelable* and *Sliceable* base classes/interfaces, we can **inject** that behavior directly into them.

Using Composition



Composition Code

```
public static void main
    new

private static void peelAndEat
    while
```

Edible Behavior

The behavior of fruits that are Edible:

```
public class EdibleBehavior implements Edible
    private double

    public EdibleBehavior
        this

    public void eat double

    public double getPercentageEaten
        return
```

Different Behaviors

Fruits can have different behaviors. One can be not to be Edible:

```
public class NotEdibleBehavior implements Edible
    public void eat double           throws
        throw new

    public double getPercentageEaten
        return
```

Orange Class

An orange is Edible, Peelable but isn't Sliceable:

```
public class Orange extends Fruit
    double
super
new
new
new
```

Problems

- Might be overly **complex** for most cases.
- There is no **type-safety**. We can only know if a *Fruit* is *Edible* in **runtime**.
- We are calling the method on the behavior instead of on the class itself.

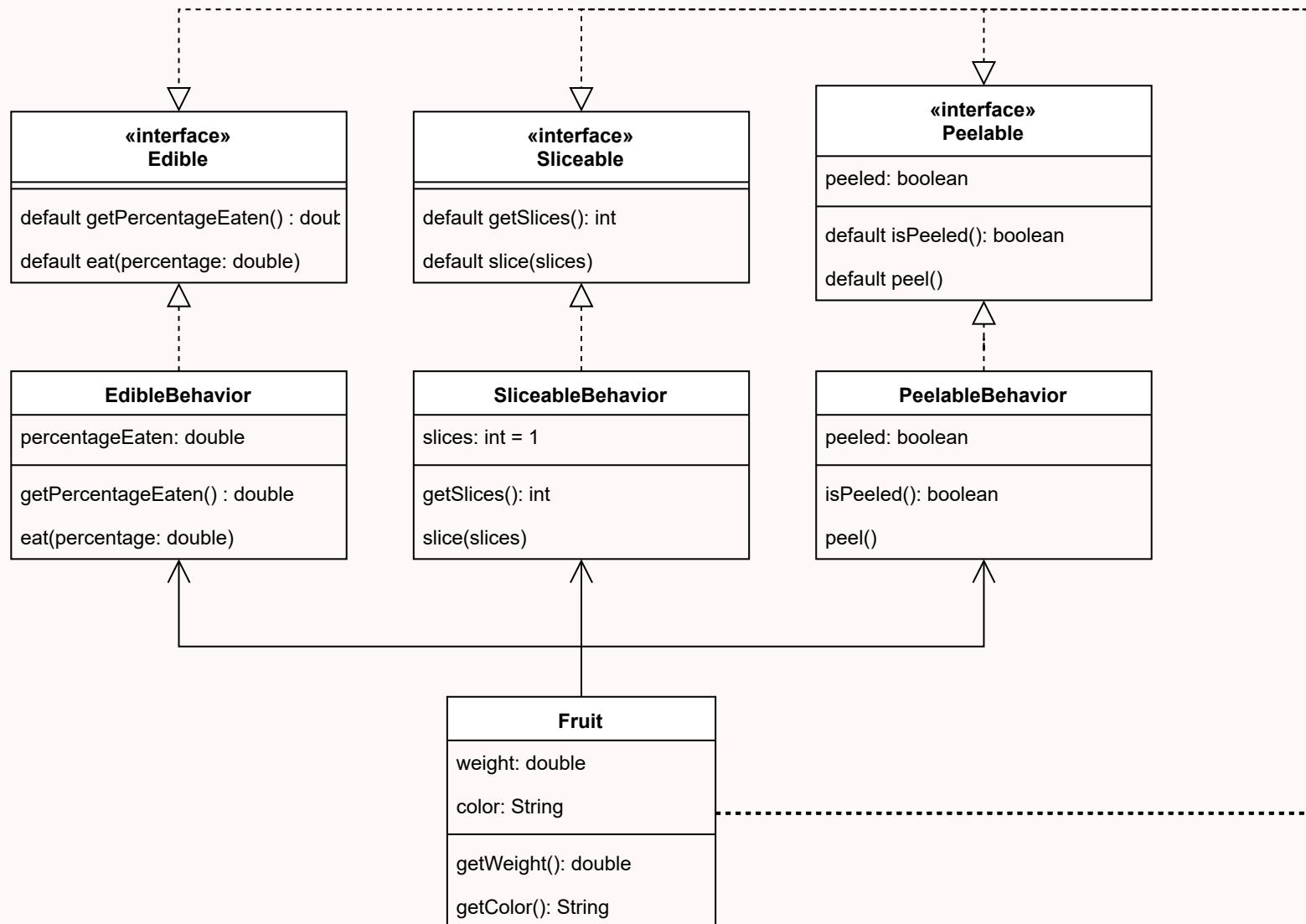
Delegation

Delegation

By having the *Fruit* class implement the different behaviors and *delegating* each method call to the corresponding one, we still get the benefits of *composition* but we now can **call the methods directly**.

All other **drawbacks** of simple composition are **still present**.

Using Delegation



Other Methods

Other Methods

- Monkey Patching (Javascript, Python, ...)
- Traits and Mixins (Scala, Ruby, ...)
- Extension Methods (C#, Kotlin, ...)
- Type Classes (Haskell, Scala, Rust)

Property-Based Testing

Using jqwik for Java

André Restivo / Hugo Sereno

Index

Testing 101

Property Based Testing

jqwik

Testing 101

The Novice Programmer

Everyone likes to reimplement stuff from scratch; in that spirit, let us code our own sum function:

```
public int mySum int  int  
int  
while  
  
return
```

How should one proceed to test it? Many will write something like this:

```
public int mySumTest
```

Everything is awesome! All tests are passing!...

The Lurking Bug

There's indeed a bug in the implementation. Look at the code very carefully:

```
public int mySum int  int  
int  
while  
  
return
```

What happens when you try something like ?

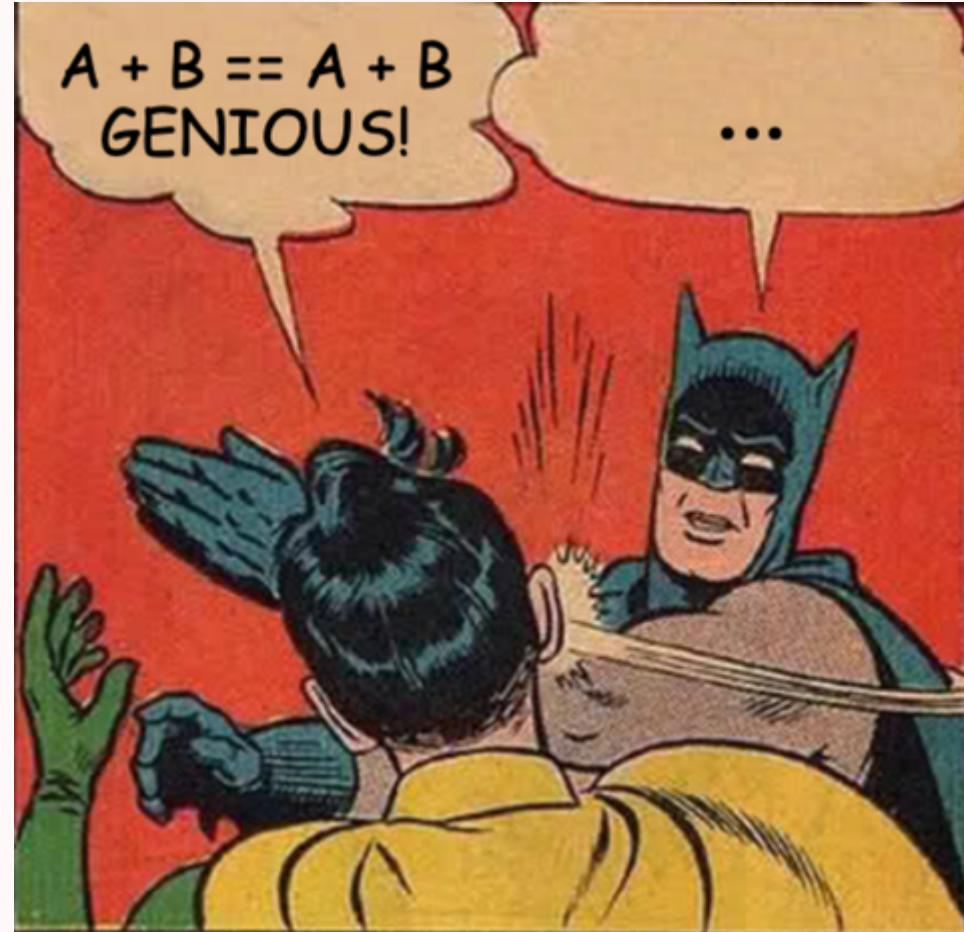
The Intermediate Programmer

But that is stupid! Why don't we use the + operator? — an older student.

```
public int mySum int  int  
return
```

... and proceed to test for a range! — the same older student.

```
public int mySumTest  
for int  
  for int
```



The problem of testing

- ... we test what we know. Because if we knew what we didn't know, we would do it right.
- ... so how can we test what we don't know?

Property-Based Testing

Property-Based Testing

... also known as the art of specifying the constraints of the outcome and asking the computer to find out if our code complies.

Let's think about sum. What can you tell us that are true for all sums without testing for a specific sum?

Property-Based Testing

... also known as the art of specifying the constraints of the outcome and asking the computer to find out if our code complies.

Let's think about sum. What can you tell us that are true for all sums without testing for a specific sum?

- Summing 0 (or to 0), is the same as doing nothing:

Property-Based Testing

... also known as the art of specifying the constraints of the outcome and asking the computer to find out if our code complies.

Let's think about sum. What can you tell us that are true for all sums without testing for a specific sum?

- Summing 0 (or to 0), is the same as doing nothing:
- Summing 1 with 0 is the same as summing 0 with 1 :

Property-Based Testing

... also known as the art of specifying the constraints of the outcome and asking the computer to find out if our code complies.

Let's think about sum. What can you tell us that are true for all sums without testing for a specific sum?

- Summing 0 (or to 0), is the same as doing nothing:
- Summing 1 with 0 is the same as summing 0 with 1 :
- Summing 1 with 1 is the same as summing 2 with 0 :

Property-Based Testing

... also known as the art of specifying the constraints of the outcome and asking the computer to find out if our code complies.

Let's think about sum. What can you tell us that are true for all sums without testing for a specific sum?

- Summing 0 (or to 0), is the same as doing nothing:
- Summing 0 with x is the same as summing x with 0 :
- Summing x with 0 is the same as summing x with 0 :
- Summing x with y and then with z is the same as:

Challenge

Suppose you don't have access to the `+` operator. How can we implement a test that uses the above properties to verify if our sum is working?

So what is PBT?

- ... usage of Arbitraries;
- ... usage of Statistics to cover the search space and provide confidence;
- ... usage of Properties to specify the external behavior of our system and search for counter-examples;

So nice things:

- ... Reproducibility (via paths and seeds);
- ... Shrinking (smallest cases that reproduce the bug).

Arbitrariness

- An **Arbitrary** is a **random generator** of a particular class (or primitive);
- If you recall discrete mathematics, it's the equivalent of saying:
 - *for a given x , where x is a natural number*
- In code, we can think of an object of ;

Arbitrariness

- An **Arbitrary** is a **random generator** of a particular class (or primitive);
- If you recall discrete mathematics, it's the equivalent of saying:
 - *for a given x , where x is a natural number*
- In code, we can think of an object of ;
- Most frameworks provide us for several **common cases**:
 - Numbers (, ...)
 - Booleans
 - Collections (, ...)

Arbitraries

- An **Arbitrary** is a **random generator** of a particular class (or primitive);
- If you recall discrete mathematics, it's the equivalent of saying:
 - *for a given x, where x is a natural number*
- In code, we can think of an object of ;
- Most frameworks provide us for several **common cases**:
 - Numbers (, ...)
 - Booleans
 - Collections (, ...)
- You can also define your own Arbitraries, either by:
 - **Mapping built-in Arbitraries**;
 - **Creating them from scratch**.

Arbitraries

- An **Arbitrary** is a **random generator** of a particular class (or primitive);
- If you recall discrete mathematics, it's the equivalent of saying:
 - *for a given x , where x is a natural number*
- In code, we can think of an object of ;
- Most frameworks provide us for several **common cases**:
 - Numbers (, ...)
 - Booleans
 - Collections (, ...)
- You can also define your own Arbitraries, either by:
 - **Mapping built-in Arbitraries**;
 - **Creating them from scratch**.
- There are things that make an Arbitrary more useful than being **merely a random generator**, which is (next slide)...

... Statistics

- The **small-scope hypothesis** claims that most inconsistent models have counterexamples within *small bounds*;
- Think about most bugs you find in code that involves integers:
 - **Zero** tends to be problematic... So does **-1** and **1**...

... Statistics

- The **small-scope hypothesis** claims that **most inconsistent models have counterexamples within *small bounds***;
- Think about most bugs you find in code that involves integers:
 - **Zero** tends to be problematic... So does **-1** and **1**...
 - -128, 129, 256, 32769, -32768, 65536... why?

... Statistics

- The **small-scope hypothesis** claims that most inconsistent models have counterexamples within *small bounds*;
- Think about most bugs you find in code that involves integers:
 - **Zero** tends to be problematic... So does **-1** and **1**...
 - **-128, 129, 256, 32769, -32768, 65536...** why?
 - And **Infinity** and **-Infinity**...
- So, while you would need to **cover all the space** to gain **perfect knowledge**, in practice, a **small number** of instances of certain Arbitrariness are responsible for most bugs;
- Can you think of 's that usually triggers buggy code?

... Statistics

- The **small-scope hypothesis** claims that **most inconsistent models have counterexamples within *small bounds***;
- Think about most bugs you find in code that involves integers:
 - **Zero** tends to be problematic... So does **-1** and **1**...
 - **-128, 129, 256, 32769, -32768, 65536...** why?
 - **And Infinity and -Infinity...**
- So, while you would need to **cover all the space** to gain **perfect knowledge**, in practice, a **small number** of instances of certain Arbitrariness are responsible for most bugs;
- Can you think of 's that usually triggers buggy code?
 - **The empty list!**

... Statistics

- The **small-scope hypothesis** claims that most inconsistent models have counterexamples within *small bounds*;
- Think about most bugs you find in code that involves integers:
 - **Zero** tends to be problematic... So does **-1** and **1**...
 - **-128, 129, 256, 32769, -32768, 65536...** why?
 - And **Infinity** and **-Infinity**...
- So, while you would need to **cover all the space** to gain **perfect knowledge**, in practice, a **small number** of instances of certain Arbitrariness are responsible for most bugs;
- Can you think of 's that usually triggers buggy code?
 - The **empty list!**
- PBT frameworks call this **biased** search, and considers it for you;
- It is up to the Arbitrary to define their **bias**.

Shrinking

Imagine testing if your *hero* can walk out of the *arena*.

If we do a **random search**, we might end up with a counter example that is similar do this:

Can we do **better** (what is better?)

Shrinking (Part II)

- **Short Answer:** Yes;
- **Long Answer:** Yes, but it's *not easy*, though the frameworks usually help.

Shrinking (Part II)

- **Short Answer:** Yes;
- **Long Answer:** Yes, but it's *not easy*, though the frameworks usually help.

The trick relies in **shrinking**. Once you've found something that produces a counter-example, you can attempt to **mutate** it in order to find a **smaller** counter-example:

- It's easier if you think about numbers. Imagine that a bug triggers with **negative numbers**; as soon as the computer finds that -5234153245 leads to an error, it can **try to shrink it**.

Shrinking (Part II)

- **Short Answer:** Yes;
- **Long Answer:** Yes, but it's *not easy*, though the frameworks usually help.

The trick relies in **shrinking**. Once you've found something that produces a counter-example, you can attempt to **mutate** it in order to find a **smaller** counter-example:

- It's easier if you think about numbers. Imagine that a bug triggers with **negative numbers**; as soon as the computer finds that -5234153245 leads to an error, it can **try to shrink it**.
- How can we **shrink a number**? What about **decreasing its magnitude**? Does $-5234153245/2 = -2617076622$ also produce a counter example? **Yes**.

Shrinking (Part II)

- **Short Answer:** Yes;
- **Long Answer:** Yes, but it's *not easy*, though the frameworks usually help.

The trick relies in **shrinking**. Once you've found something that produces a **counter-example**, you can attempt to **mutate** it in order to find a **smaller** counter-example:

- It's easier if you think about numbers. Imagine that a bug triggers with **negative numbers**; as soon as the computer finds that -5234153245 leads to an error, it can **try to shrink it**.
- How can we **shrink a number**? What about **decreasing its magnitude**? Does $-5234153245/2 = -2617076622$ also produce a counter example? **Yes**.
- If you apply **shrinking strategies recursively**, you'll eventually reach -1 . Which is a much nicer counter-example to deal with :)

Shrinking (Part II)

- **Short Answer:** Yes;
- **Long Answer:** Yes, but it's *not easy*, though the frameworks usually help.

The trick relies in **shrinking**. Once you've found something that produces a counter-example, you can attempt to **mutate** it in order to find a **smaller** counter-example:

- It's easier if you think about numbers. Imagine that a bug triggers with **negative numbers**; as soon as the computer finds that -5234153245 leads to an error, it can **try to shrink it**.
- How can we **shrink a number**? What about **decreasing its magnitude**? Does $-5234153245/2 = -2617076622$ also produce a counter example? **Yes**.
- If you apply **shrinking strategies recursively**, you'll eventually reach -1 . Which is a much nicer counter-example to deal with :)

Can you think of **strategies** to shrink an

?

jqwik

jqwik

The main purpose of **jqwik** is to bring **Property-Based Testing (PBT)** to the **JVM**.

A **property** is supposed to describe a **generic invariant or post-condition** of your code, given some **precondition**.

jqwik will then try to generate many value sets that fulfill the precondition hoping that one of the generated sets can falsify a **wrong assumption**.

[jqwik documentation](#)

Gradle

To use **jqwик**, you just need to add the following to your build.gradle:

Notice that we need to use JUnit5 instead of JUnit4.

Properties

To define a **property**, we just need to add the **annotation** to our test:

```
class TestNumbers

public void testSumAssociativity      int      int      int
    assert
```

Notice that our test receives **three parameters** (a, b, and c) and that we are saying that the property should **hold for all** possible values using the **annotation**.

Parameter Generation

jqwik is capable of generating parameters for a **wide range of types**: Strings, all kinds of numerical types, booleans, characters, Lists, Sets, Streams, Arrays, ...

Here we are testing if reversing any integer list twice results in the same list:

```
public void testDoubleReverse
    assert

public          reverseList
    new

for

return
```

Constraining Parameters

Sometimes we want to constrain the generated parameters. For example, the following test:

```
public void testDivision      int
```

Does not work if the number is zero, so we can use the
number:

annotation to constrain the

```
public void testDivision      int
```

Maybe this should also work for negative numbers!

Constraining Parameters

There are several types of constraints that can be applied:

- If we want to generate null values, value is the percentage of null values to generate.
- Prevents repeated values in a list.
- A fixed size string or between min and max characters.
- , , , , , , , Several ways to constraint character generation.
- , , To constraint the size of generated lists.
- , , , , , , , ... To constraint generated numbers.

Constraining Parameterized Types

If we want to constrain the generation of **contained parameter types** we can annotate the parameter type **directly**:

```
public void testListSumPositive
    assert

private int sum
    int
    for int
    return
```

This property does not hold, why?

Arbitrary

If the **default generators** are **not enough**, we can use the **create new generators**:

Fixing the **test:**

```
public void testDivision(int dividend, int divisor) {
    assertEquals(0, division(dividend, divisor));
    assertEquals(1, division(1, 1));
    assertEquals(1, division(1, 2));
    assertEquals(-1, division(-1, 1));
    assertEquals(1, division(1, -1));
    assertEquals(-1, division(-1, -1));
    assertEquals(1, division(0, 1));
    assertEquals(0, division(1, 0));
    assertEquals(0, division(0, 0));
}
```

With arbitraries we can generate **integers**, **floats**, **strings**, ... which we can then restrict using functions like **between**, **greaterThan**, **lessThan**, **notNull**, **notEmpty**, **notZero**, ...

Combining Arbitraries

We can even combine *arbitraries* using the lambda magic:

```
void testWithPlates  
  
    carPlates  
return
```

and methods, sprinkled with some Java

Another Arbitrary Example

A prime number cannot be divided by any(?) other number:

```
primeNumbers

return

private boolean isPrime
for int
if      return false
return true

void testWithPrimes
int
ForAll      int
assert
```

Output

The result of a test in jqwик looks something like these:

In this report we can see the number of test runs for this property (), number of calls that were not rejected (), how values were generated (), if we should keep using the same seed if a property check fails (), and which seed was used ().

Configuring Runs

We can change some configuration parameters for each test:

```
public void testDoubleReverse  
assert
```

Shrinking

The advantage of using arbitraries instead of just using random data generators, is that arbitraries know how to **shrink**:

```
public void testDifferenceAssociativity
    int
    int
    int
assert
```

This allows us to find smaller examples that are easier to understand.

An Hero example...

Testing if the arena bounds are correctly checked:

```
public void testArenaBounds
    IntRange           int
    int               int
    int               int
    new              null

assert      new
assert      new
assert      new
assert      new
```

...or two!

Testing if the hero never leaves the arena:

```
public void testMovingBounds
throws

    new      new
    new
        new
    new

while

assert
assert

assert
assert
```

Mocking for PBT

In this second example we had to create a specialized **mock** for the class:

```
public class ArenaViewMock extends ArenaView
    private

    public ArenaViewMock
        throws
        super null null
        this

    public      getAction  throws

    return

    public void draw  throws

    public boolean hasMoreActions  return
```