Software Engineering

Lecture 02 – Git & OOP

© 2015-20 Dr. Florian Echtler
Bauhaus-Universität Weimar
<florian.echtler@uni-weimar.de>

Revision Control

- Also known as version or source code control
- Revision control systems (RCS) maintain …
  - a history of changes
  - from multiple persons
  - to a set of documents.
- Mostly designed for plain-text documents (e.g. source code, LaTeX files, …)
- Extensions for binary files (e.g. images) possible

Storing history in RCS (1)

- Most RCS use numbered revisions
- Initial state of the document set is revision 1,
- First change is revision 2, second change ...

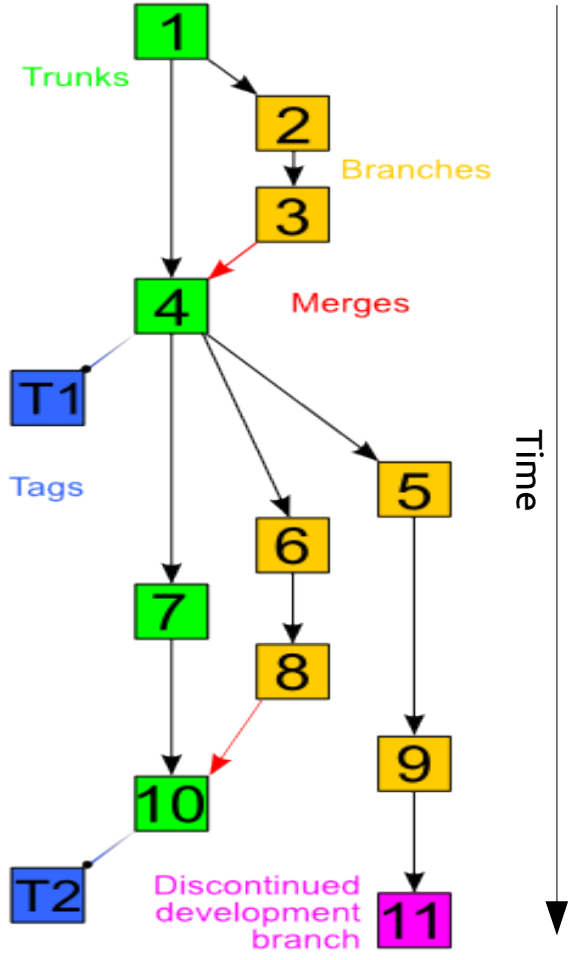
- Important: not necessarily linear
- Revisions can have multiple successors

  → overall structure is a tree (with exceptions)

Storing history in RCS (2)

Image source (CC): https://en.wikipedia.org/wiki/Revision_control

- Trunk: main development history (often also called master)
- Branch: e.g. development of extra features, bugfixes, …
- Tags: “bookmarks”, e.g. releases
- Merges: combination of 2 or more branches (break pure tree structure)

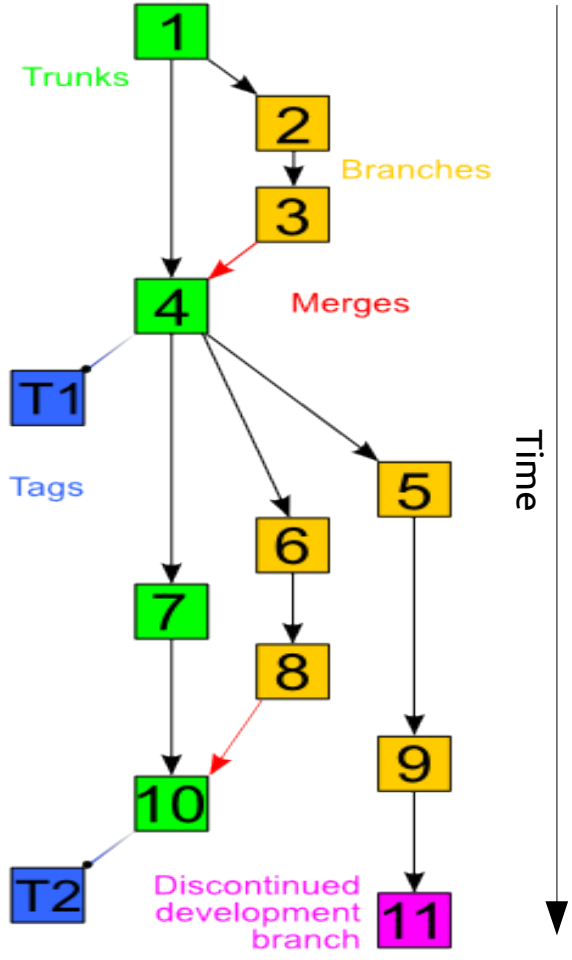


Storing history in RCS (3)

Image source (CC): https://en.wikipedia.org/wiki/Revision_control

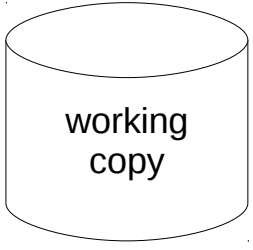
- Merges: can lead to conflicts
- e.g. what if change sets 6 and 7 edit the same file?
- What if it's the same line?

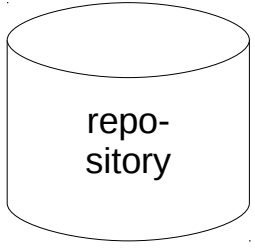
  → may require manual intervention/rewriting



Common RCS terms

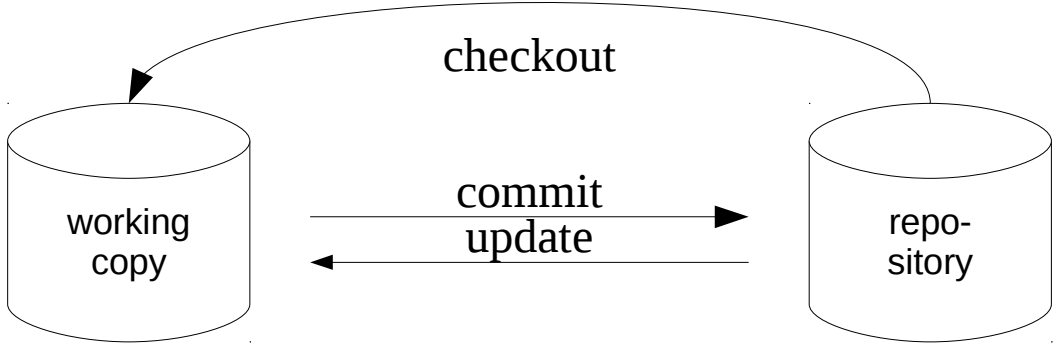
- Repository (repo): storage for files + history (usually on a remote server)
- Working copy: local copy of the files at a specific revision
- Common examples: CVS, Subversion (SVN)





Common RCS operations

- checkout: create a local copy
- commit: push a set of changes to the repository (atomic operation)
- update: integrate new changes from repo into local copy (possibly requiring merge)

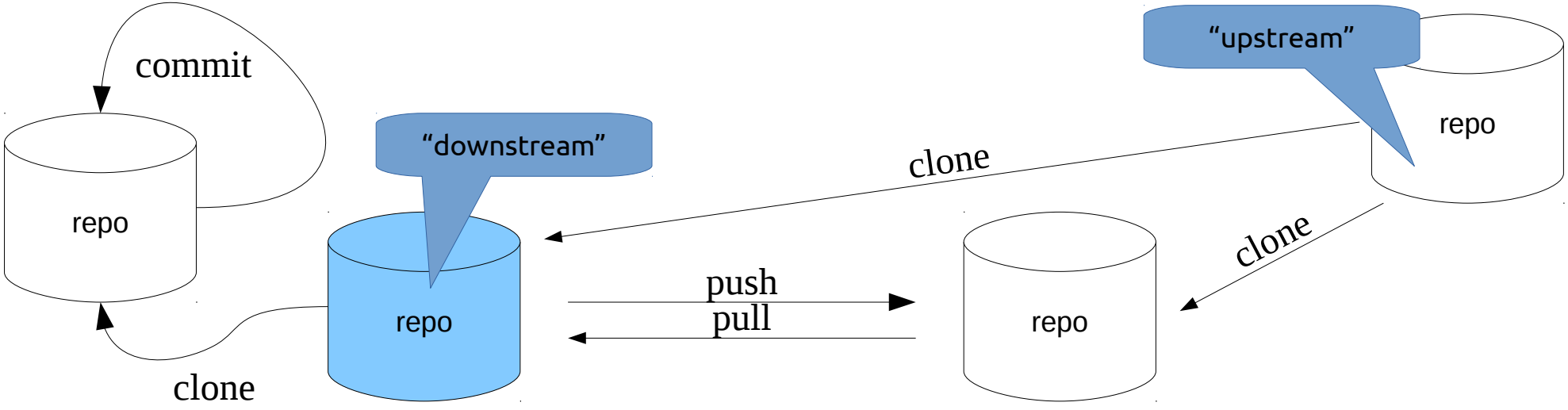


Distributed RCS

- multiple interconnected repositories (peer-to-peer), no separate working copies
- e.g. Bazaar (bzr), Mercurial, git
- widely used in open-source context

DRCS operations

- clone: create new, complete copy of repo
- commit: save changes *locally* to history
- push/pull: transfer to/from remote repo



Other RCS ops: diff

- diff: view highlighted set of changes
- +/- represents added/removed lines
- optionally also with changes per word
- works best for text docs, e.g. source code

```
diff --git a/examples/protonect/src/libfreenect2.cpp b/examples/protonect/src/libfreenect2.cpp
index 2d4709b..42e2157 100644
--- a/examples/protonect/src/libfreenect2.cpp
+++ b/examples/protonect/src/libfreenect2.cpp
@@ -422,7 +422,8 @@ bool Freenect2DeviceImpl::open()
   if(usb_control_.setVideoTransferFunctionState(UsbControl::Disabled) != UsbControl::Success) ...

-  size_t max_iso_packet_size = libusb_get_max_iso_packet_size(usb_device_, 0x84);
+  int max_iso_packet_size;
+  if(usb_control_.getIrMaxIsoPacketSize(max_iso_packet_size) != UsbControl::Success) return false;

...
```

RCS best practices

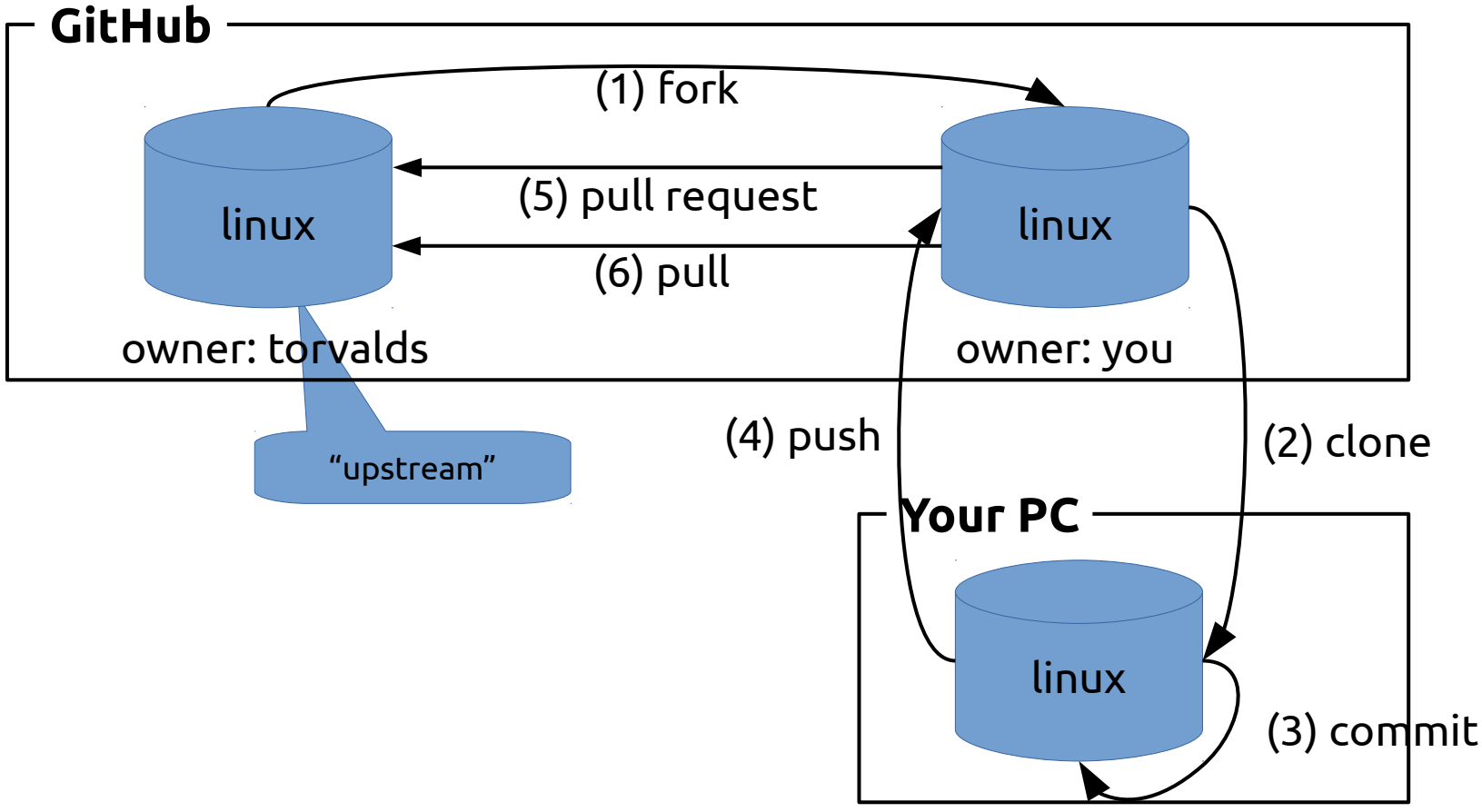
- Keep commits small
- Only related changes in one commit
- Use meaningful commit messages
- *Only* commit valid code (at least compiles)
- RCS is not a backup
- Use graphical tools

https://xkcd.com/1296/

General best practices: Workflow

- A common git workflow with Github:
  - Pull recent changes from upstream repository.
  - Check for new issues.
  - Create and checkout a new branch.
  - Fix an issue in this branch.
  - Test and commit the branch locally.
  - Push the new branch to your Github repository.
  - Create a pull request for the master repository.

General best practices: Workflow



Object-Oriented Programming

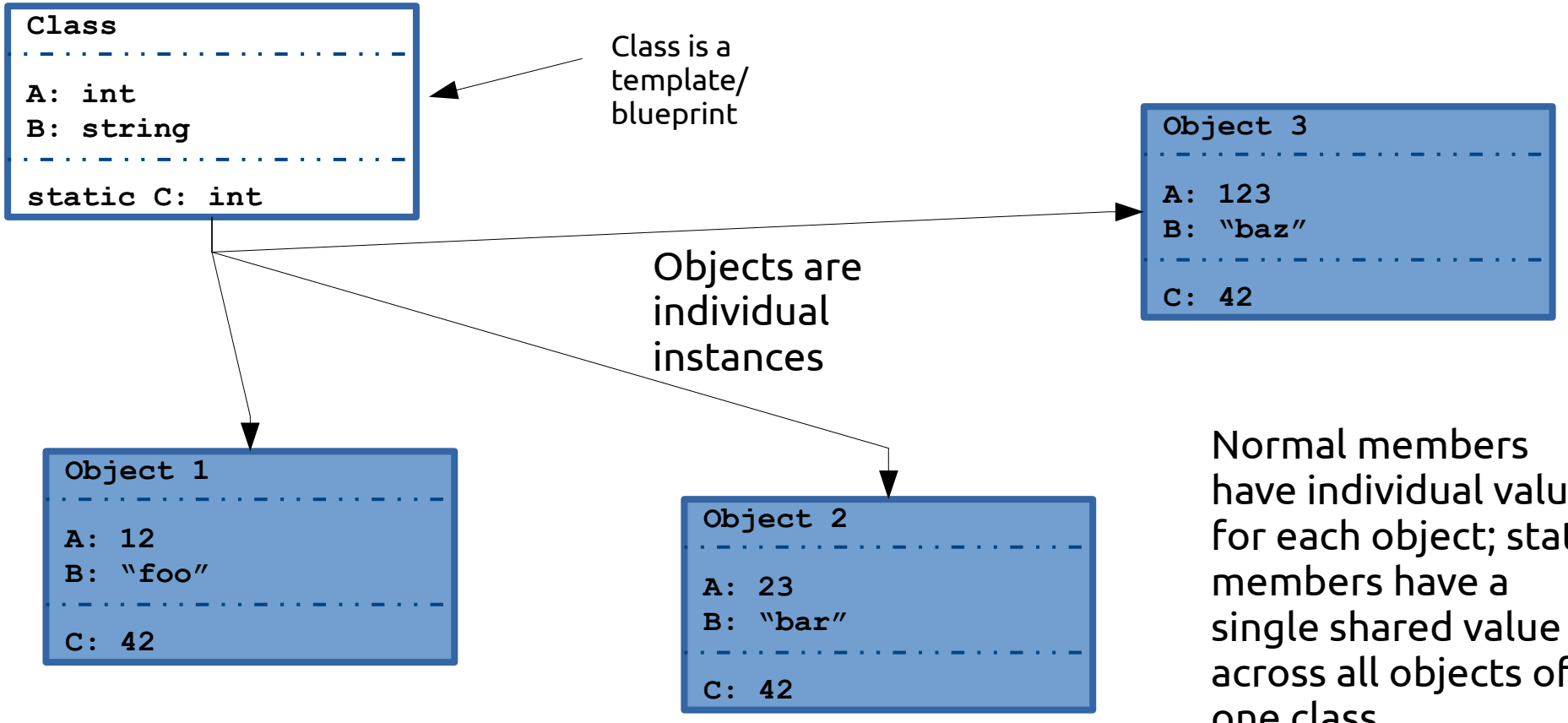
- Classes and Objects
- Encapsulation
- Inheritance and Polymorphism
- Object-Oriented Design

Classes and Objects (1)

- A *class* is a template with …
  - *Variables* (placeholders for data)
  - *Methods* (manipulators for data)

- An *object* is a single instance of that class
  - Concrete values for variables
  - Many objects of same class can coexist
  - Special method (*constructor*) for initialization

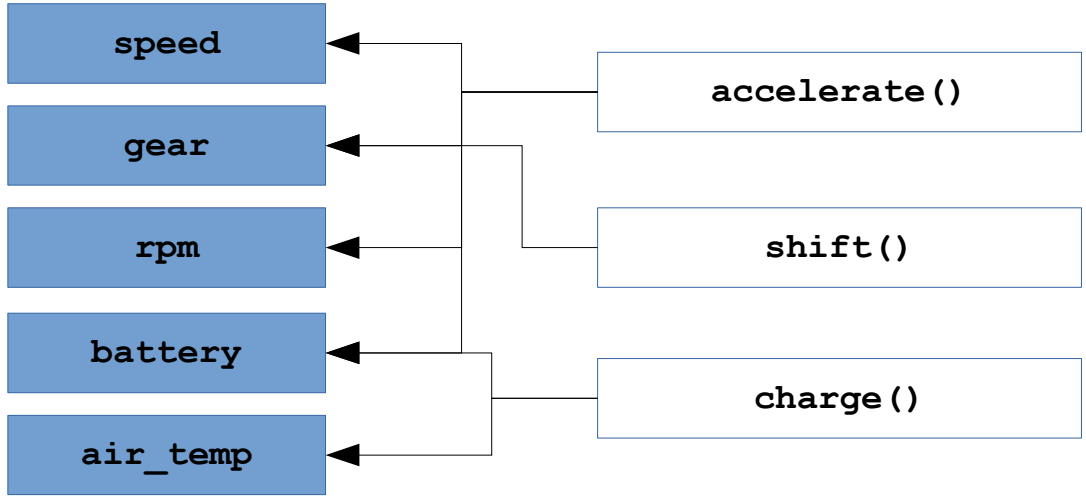
Classes and Objects (2)



Normal members have individual values for each object; static members have a single shared value across all objects of one class.

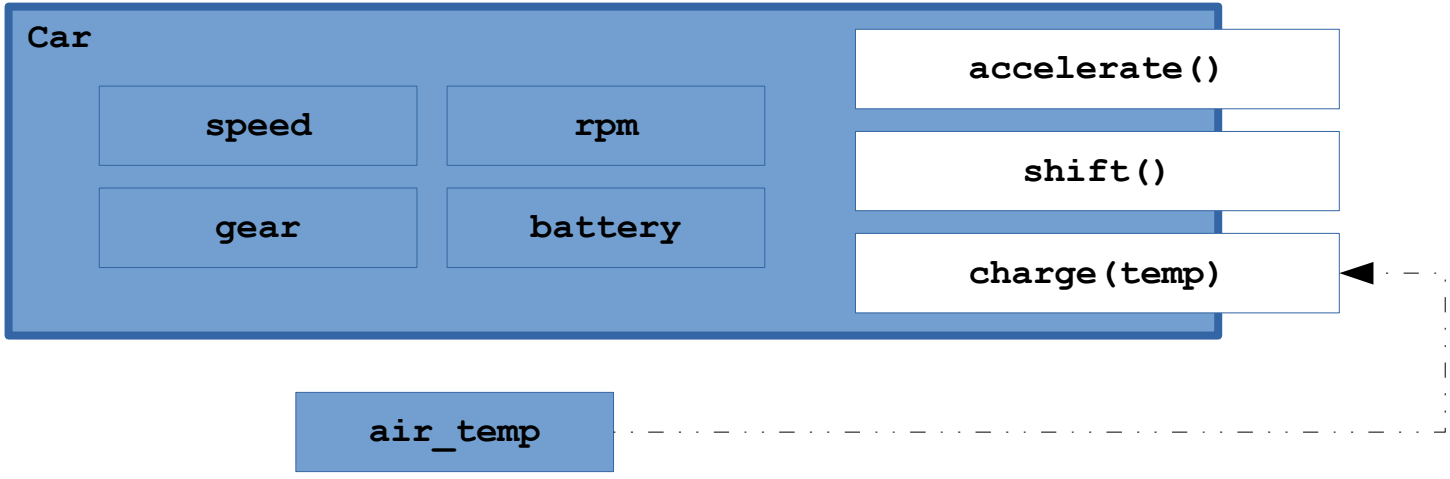
Encapsulation (1)

- Procedural programming: global state (variables), modified through functions
- Hard to keep track of side effects (cf. Toyota)



Encapsulation (2)

- Core idea of OOP: *encapsulate* related data
- Data is no longer directly accessible
- Class provides *methods* to manipulate data



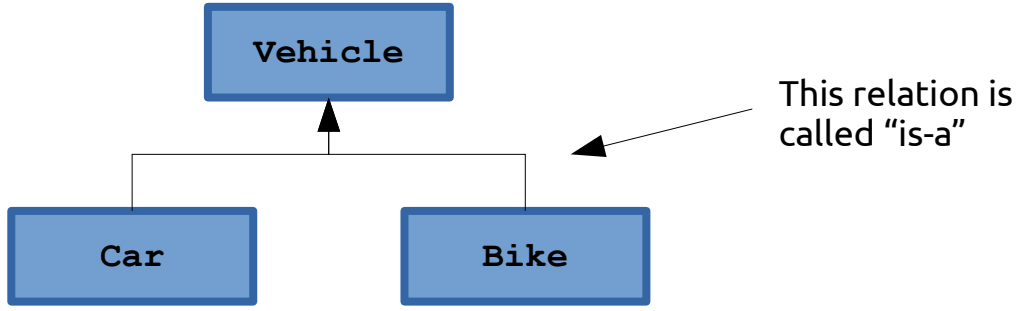
Encapsulation (3)

- Data and methods have *visibility* or *scope*
- Common levels:
  - Public: visible/accessible to everyone
  - Protected: only visible to subclasses (see below)
  - Package (default in Java): visible to package
  - Private: only visible to other class members

- Rule of thumb: try to avoid public members (otherwise, encapsulation is sidestepped)

Inheritance

- Classes can be derived from other classes
- Superclass (parent) → Subclass (child)
- Class members are *inherited* from superclass
- Subclasses often introduce extra variables/methods (specialization)



Polymorphism

- Subclasses can overwrite behaviour
- Method with same *signature* as in superclass
  - Signature = name + parameter types
  - Actual runtime behaviour depends on subclass
- e.g. `Vehicle` has `shift()` method
  - `Bike.shift()/Car.shift()` behave differently
  - Possible to call `shift()` on any `Vehicle`
  - `Car/Bike` can always be *upcast* to `Vehicle`

Generics

- Can be used to create class “families”
- E.g. Container class:
  - `public class Container<T> { … }`
  - Can be used as `Container<String>`, `Container<Int>`, `Container<Other>`, …
- Advantages:
  - Less repetitive code (one container for all types)
  - Less runtime errors (`Container<String>` can only ever contain `String` objects)

Object-Oriented Design

- Goal: modularize a system specification
- Decompose/subdivide system by objects which are (supposed to be) manipulated
- Question: how to find these objects?
  - Data-driven design
  - Responsibility-driven design

- Important: iterative process!

Data-Driven Design (1)

- Focus on the data an object contains
  - E.g. `Student` class has `name`, `id`, `courses`, …
  - `Course` has `name`, `teacher`, `requirements`, …
- Classes (usually) match real-world objects
- Subclasses match real-world categories
  - E.g. `Student` is a subclass of `Person`,
    `Lecture` is a subclass of `Course`, …

Data-Driven Design (2)

- Problem: where to put actual “business logic”?
  - Usually solved by central “manager” class
  - (Somewhat) contrary to core OOP concepts
- Mostly suited for database-like apps (e.g. store inventory, university management, …)

Responsibility-Driven Design

- Focus on the functions an object performs
  - Find *candidate classes* in system architecture
  - Determine *responsibilities* of each class
  - Determine *collaboration* between objects

- Classes have less connection to real world, e.g. `OrderProcessor`, `CourseCatalogue`
- Subclasses now reflect common *behaviour*

RDD – Finding Candidate Classes

- Candidate classes come from nouns …
  - in the system specification
  - during discussion
  - in background knowledge

RDD – Finding Candidate Classes

Source (FU): https://www2.cs.arizona.edu/~mercer/Presentations/OOPD/12-RDD-Jukebox.pdf

The student council wants to install a Jukebox in the student center. The Jukebox must allow students to play a song. No money will be required. Instead, a student will swipe an ID card through a card reader, view the song collection and choose a song. Students will each be allowed to play up to 1500 minutes worth of "free" Jukebox music in their academic careers, but never more than two songs on any given date. No song can be played more than five times a day.

RDD – Finding Candidate Classes

- Candidates:
  - From spec: student council, jukebox, student center, students, song, money, ID card, card reader, song collection, two songs, 1500 minutes, music, academic career, date, five times a day.
  - From context: stereo, amplifier, speaker?

RDD – Finding Candidate Classes

- Guidelines:
  - One word for one concept: song, music → `Song`; students, ID card → `Account`
  - Model values of attributes, not attributes themselves: 1500 minutes, two songs, 5 times per day → attributes of `Song` / `Account`
  - Be wary of adjectives: not applicable here, usually also attributes instead of separate classes
  - Focus on the problem domain: student council, student center, money, speaker, … → not applicable

RDD – Determine Responsibilities

- Responsibilities are:
  - The knowledge a class maintains/provides
  - The actions it can perform

- Responsibilities == “public services”
  - Basic client-server approach
- Every class can be …
  - A client, using services of other classes
  - A server, providing services to other classes

RDD – Determine Responsibilities

- E.g. responsibilities of `Song`:
  - `Play` (but only max. 5 times a day)
- `Account`:
  - `ChooseSong` (but only 2 times a day)
- `Jukebox`:
  - `Login` (if confirmed by ID card)

RDD – Determine Collaborations

- Which class needs which other service?
  - `Jukebox` may need `AccountManager`,
  - `Song` may need `DateTime` & `Database`, ...

- Collaborations reveal control/data flow
- Collaborations can uncover missing functions

Questions/suggestions?