# EECS3311- Software Design

Quality First

# Devops 2019

Despite all the great methodologies we have in IT, delivering a project to production still feels like going to war. Developers are nervous because they have the pressure of delivering new functionality to the customer as fast as possible. On the other side, operations resists making that change a reality because it knows change is a major cause of outages. So the usual finger-pointing begins when problems arise: "It's a development problem"; "Oh no, it's an operations problem." This tragic scene gets repeated over and over again in many companies, much to the frustration of management and the business, which is not able to predict releases and deliver business value as expected and required.

# HackerRank 2019

## One of the biggest bugs: ‘Wiped out database’

When asked about their biggest bug in production, deploying untested or broken code was the most common response. We also found that ~10% of developers admitted to wiping out the entire database.

### The biggest bug in production

# The #1 pet peeve: Badly written documentation

HackerRank 2019

Developers' struggles are real...and we wanted to find out what frustrated them the most. Junior developers particularly despised badly written documentation while senior developers thought spaghetti code was the worst.

## The #1 pet peeve at work

# Cost of bug fixes >= $300 billion/year

## The cost of software bugs could be huge

**2014 Heartbleed**
**>$500 Million**

knight CAPITAL

**2012 Logical error**
**$10 Million per minute**

MT.GOX

**2011 Bitcoin Hack**
**lost 800K bitcoins**

**1996 Ariane-5 crashed for an overflow error**
**>$7 Billion**

# Node/Javascript left_pad fiasco

However, it is not all good news. There are many cases where code reuse has had negative effects, leading to an increase in maintenance costs and even legal action [2, 29, 35, 41]. For example, in a recent incident code reuse of a Node.js package called left-pad, which was used by Babel, caused interruptions to some of the largest Internet sites, e.g., Facebook, Netflix, and Airbnb. Many referred to the incident as the case that 'almost broke the Internet' [33, 45]. That incident lead to many heated discussions about code reuse, sparked by David Haney's blog post: *"Have We Forgotten How to Program?"* [26].

While the real reason for the left-pad incident was that *npm* allowed authors to unpublish packages (a problem which has been resolved [40]), it raised awareness of the broader issue of taking on dependencies for trivial tasks that can be easily implemented [26].


Code

```
1   module.exports = leftpad;
2   function leftpad (str, len, ch) {
3     str = String(str);
4     var i = -1;
5     if (!ch && ch !== 0) ch = ' ';
6     len = len - str.length;
7     while (++i < len) {
8       str = ch + str;
9     }
10    return str;
11  }
```

Code

```
1   module.exports = leftpad;
2   function leftpad (str, len, ch) {
3     str = String(str);
4     var i = -1;
```

```
) ch = ' ';
gth;
```

What concerns me here is that *so many packages and projects* took on a **dependency** for a simple left padding string function, rather than their developers taking 2 minutes to write such a basic function themselves.

As a result of learning about the left-pad disaster, I started investigating the NPM ecosystem. Here are some of the things that I observed:

- There's a package called isArray that has 880,000 downloads a day, and 18 million downloads in February of 2016. It has **72 dependent NPM packages**. Here's its **entire 1 line of code**:

  ```
  return toString.call(arr) == '[object Array]';
  ```

Code

```
1   module.exports = leftpad;
2   function leftpad (str, len, ch) {
3     str = String(str);
4     var i = -1;
5     if (!ch && ch !== 0) ch = ' ';
6     len = len - str.length;
7     while (++i < len) {
8       str = ch + str;
9     }
10    return str;
11  }
```

**In the original**
**No comments**
**No tests**

**Wrong outputs**:

```
console.log(leftpad ('abc', 5, 345))
  // result: 345345abc

str = new Foo();
console.log(leftpad (str, 25, 'x'))
  // result: xxxxxxxxxx[object
Object]
```

# Specification/Documentation

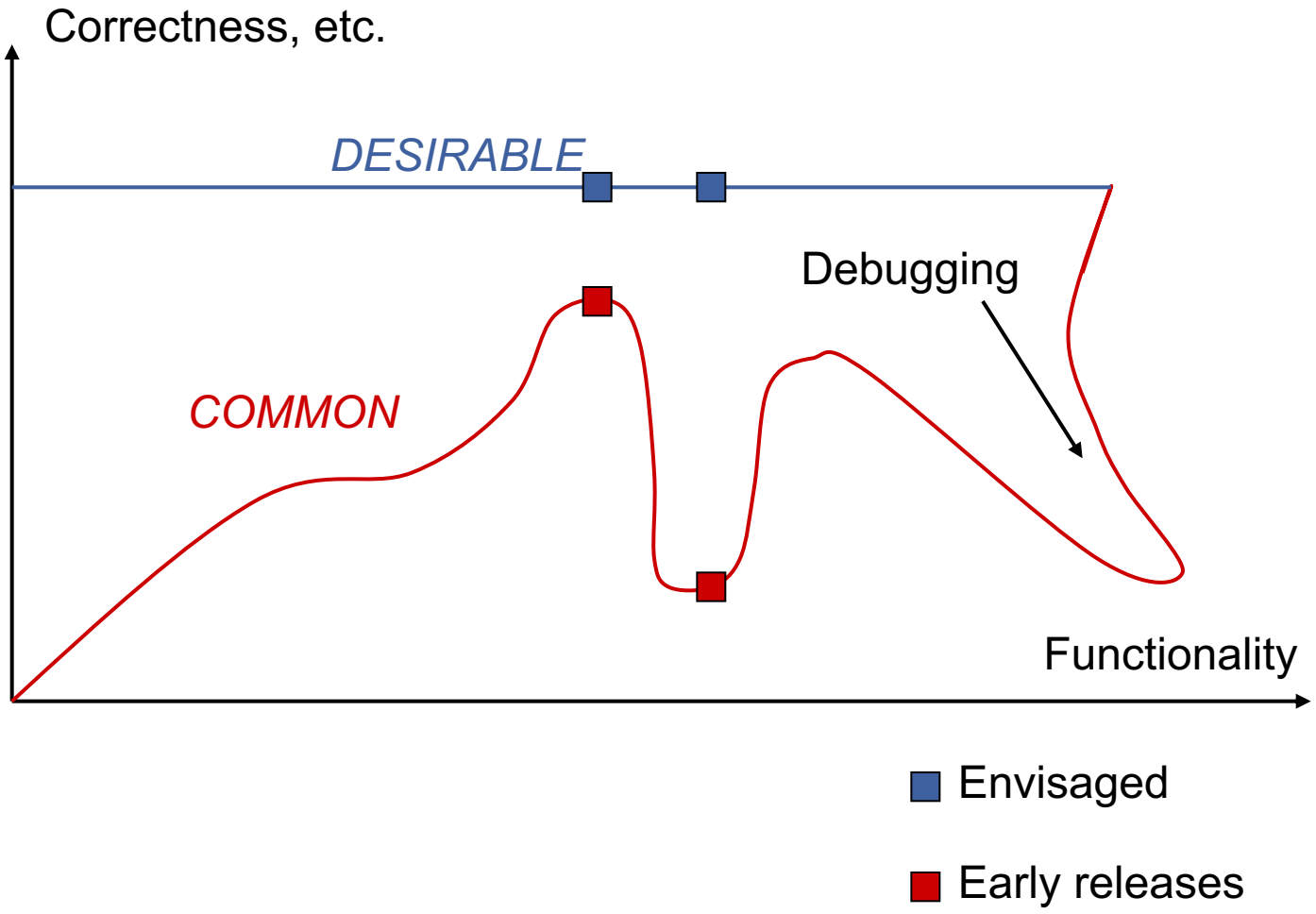
```
left_pad (str: STRING; len: INTEGER; ch: CHARACTER): STRING
                -- return a string of length `len`
                -- that is padded to the left with `len - str.count` characters `ch`
                -- retaining for the rest the original string `str`
                -- Time complexity of this feature is O(logn)
        require
                len > str.count
        ensure
                correct_length: Result.count = len
                original_string_not_changed: str = old str
                padding_correct:
                   -- ∀i ∈ 1..(len - #str): Result[i] = ch
                original_correct:
                   -- ∀i ∈ (len - #str + 1)..len: Result[i] = str [i - (len - #str)]

        end
```

# Osmond Principle

- Software Quality Includes
  - Functionality (how much the software does)
  - Everything else (correctness, robustness, efficiency, usability etc.)

# Quality goals: the Osmond curves



# Can we ship?

- In the slopy model, you may not be able to sleep at night (it may crash)
- In the flat model, the only question to ask is "does it do enough to impress the marketplace"?

- Principle: Quality Engineering will beat RAD in the long run.

# Quality First

- Principle: If the existing functionality does not yet work perfectly, don't move on to the next function — make things right first
- This will involve using
  - Design by Contract (DbC),
  - Regression Unit Testing (Espec) Seamless intertwining of analysis, design and coding
  - Regression Acceptance Testing (ETF)

# Regression Testing/ETF

Using Regression Testing, the debugger (and setting the Execution Parameters) the debug/fix exercise should be relatively straightforward.

- Add the new test (e.g. at12.txt) to the Regression Tester
- See all the old tests succeed and the new one fail
- Set up the Execution Parameters to point to at12.txt (See Run --> Execution Parameters)
- Set up the break point in the debugger to catch the failing case, e.g. at ETF_COMMAND.
- Fix it
- Rerun all the regression tests and see them all succeed.

# Quality First

- Make sure each feature works before going to the next (Unit-Test/DbC/Acceptance-test).
- Always have a working (executable) product of perfect quality - the main quality criteria are reliability and efficiency.

# Quality First

- Although at any moment the functionality is incomplete,  the functions that are implemented are tops - the functionality that is implemented could be shipped today.
- “Can we ship now?” becomes “Does it do enough to impress the marketplace?”.
- If you decide to ship an early version you will usually be able to sleep at night - it won’t crash.

# Quality First

- Get the cosmetics right before anything else - apply style rules, indexing clauses, comments, good name choices, even punctuation [OOSC2 Ch. 26 - a sense of style].
  - It is tempting to cut corners and postpone writing indexing clauses or comments, but
  - I  censure myself because I know that in the end it will mean slower progress;
  - Self documentation.

# Quality First

- Recompile all the time (get nervous if you last recompiled 5 minutes ago; continually run your unit tests).
  - If you go too long without a recompile you will soon spend hours just getting your programs to compile.
  - Compilation alone catches a large variety of errors e.g. through strong type checking
  - Refactor to get rid of code "smells" e.g. code redundancy, e.g. apply Single Choice Principle
  - Each class a single abstraction (OOSC2 – chapter 22 – How to find the classes)
- Execute right away (unit tests?) with assertion checking turned on

## Class Consistency principle

All the features of a class must pertain to a single, well-identified abstraction.

- There is a clearly associated abstraction, which can be described as a data abstraction (or as an abstract machine).
- The class name is a noun or adjective, adequately characterizing the abstraction.
- The class represents a set of possible run-time objects, its instances. (Some classes are meant to have only one instance during an execution; that is acceptable too.)
- Several queries are available to find out properties of an instance.
- Several commands are available to change the state of an instance. (In some cases, there are no commands but instead functions producing other objects of the same type, as with the operations on integers; that is acceptable too.)
- Abstract properties can be stated, informally or (preferably) formally, describing: how the results of the various queries relate to each other (this will yield the invariant); under what conditions features are applicable (preconditions); how command execution affects query results (postconditions).

Look at the Contract View

# Quality First

- Intertwine analysis, design and coding.
- Use BON static and dynamic diagrams where necessary
- Transform everything into program text as early as possible, program text that also captures the Design

# Quality First

- Include assertions (DbC) all over the place and have assertion checking turned on when you execute.
  - This catches errors not caught by type checking
- Take care of abnormal cases right away.
- Always have a working system that runs all your tests.

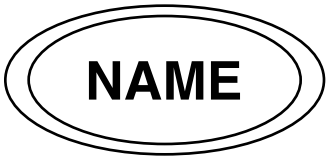
# Use BON correctly

# Classes (Compressed)

Use to draw views with lots of classes

- bird's eye view
- early stages of design


Parameterized



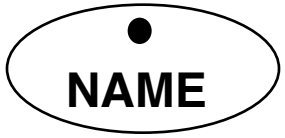

Root
Instances may be separate processes


Shortest form


Reused library


Deferred


Implemented

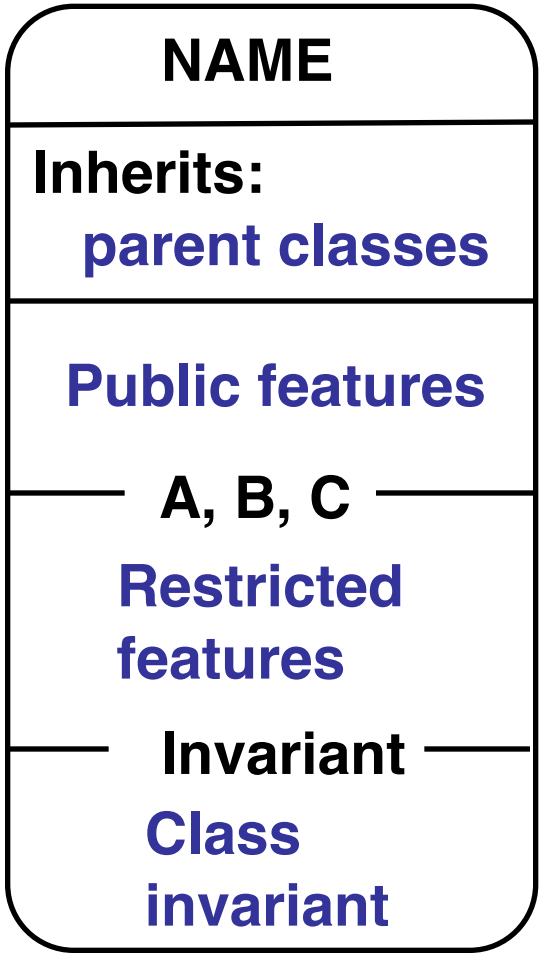



Persistent


Interfaces with outside world

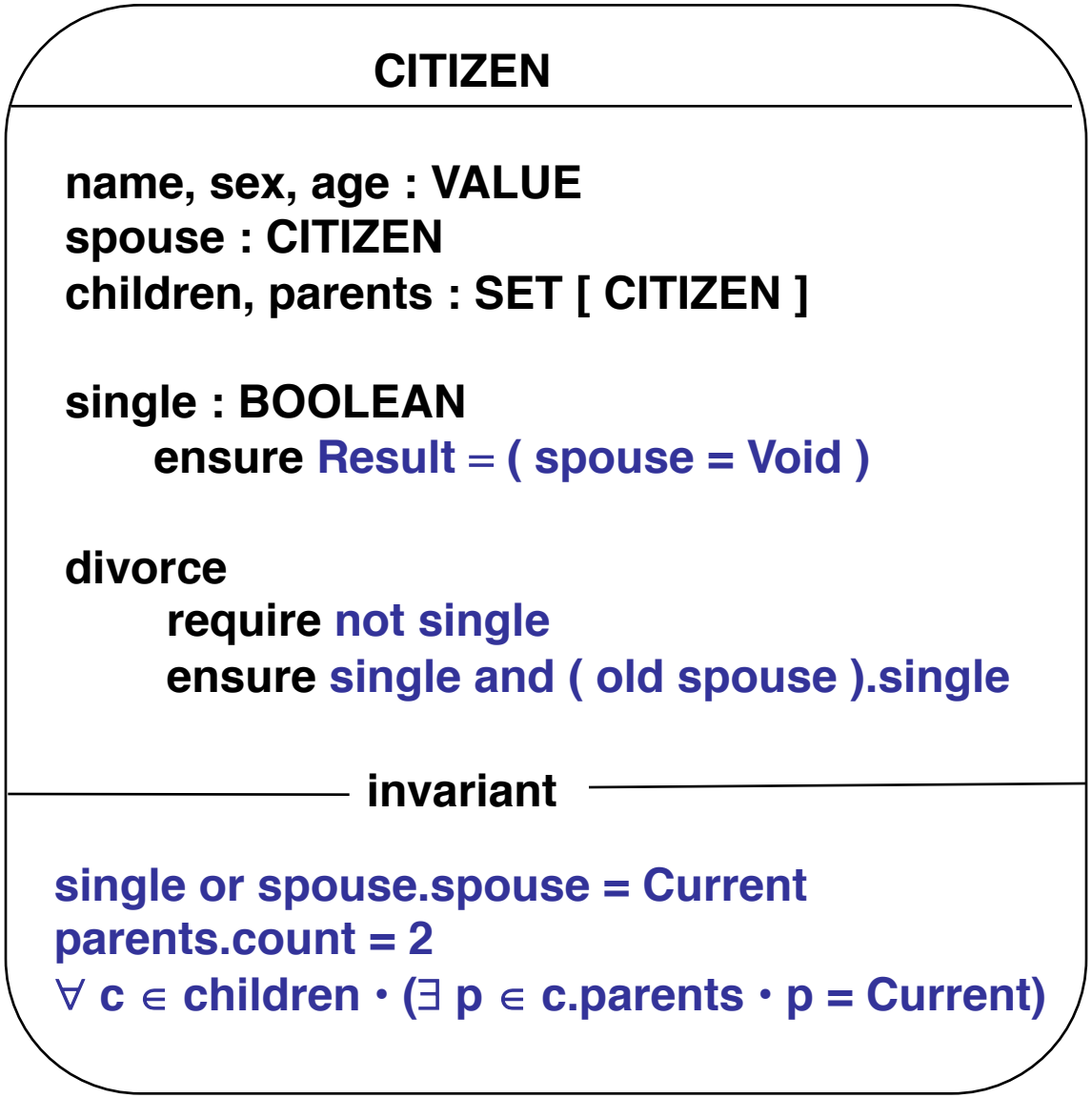
# Interface/Export policy

- Early phases concentrate on public features
- Restricted features produced during detail design
- Arbitrary number of sections, each with export list
- Each feature has a signature and optionally a behavioural specification
- Conventions
  - Classes all in upper case
  - features all in lower case
  - use underscore for longer names

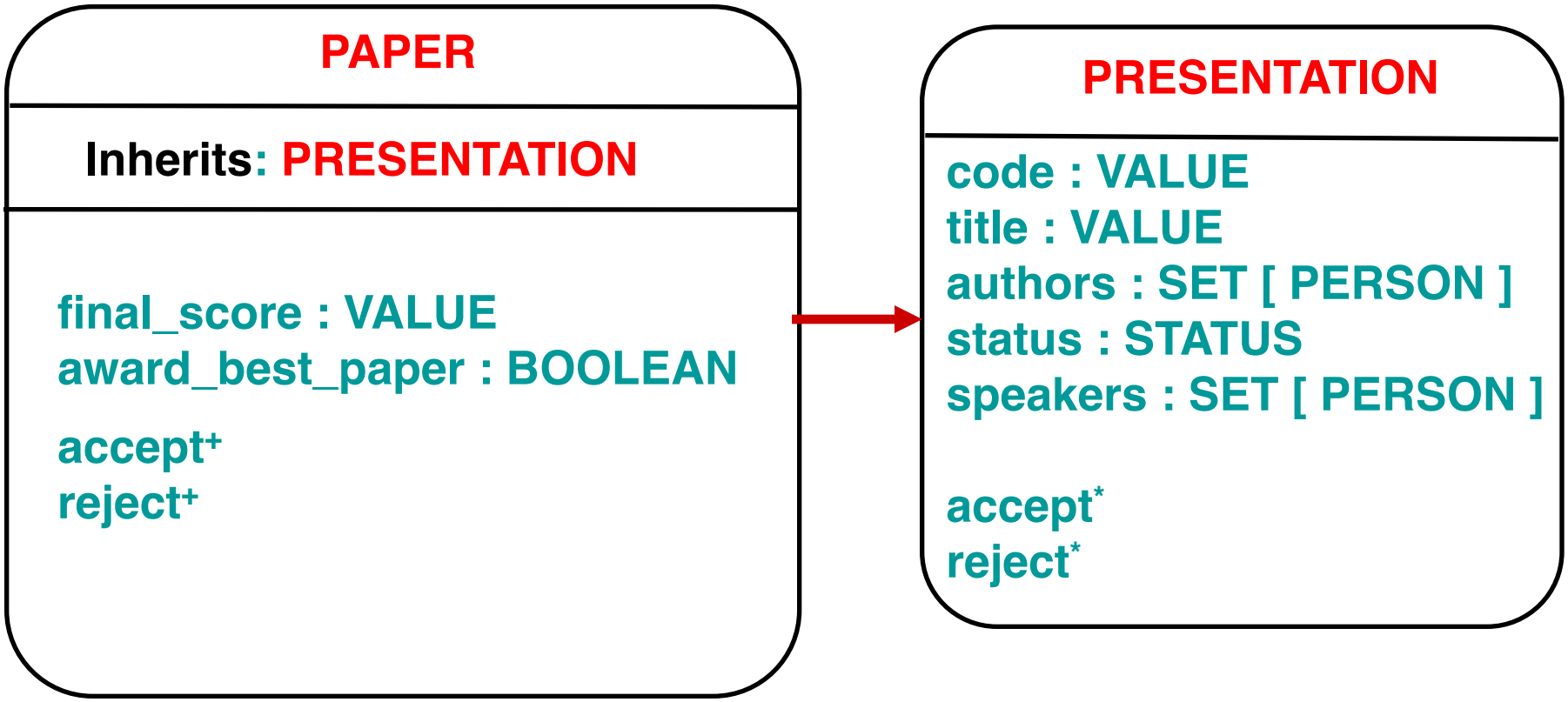


# Graphical BON Class (Uncompressed)

No need to show all features, just those of interest for the view

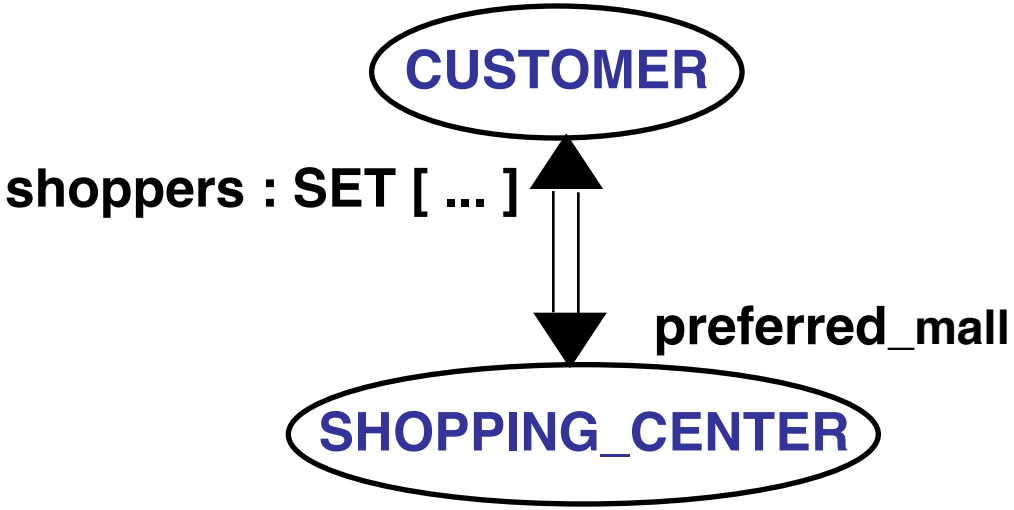


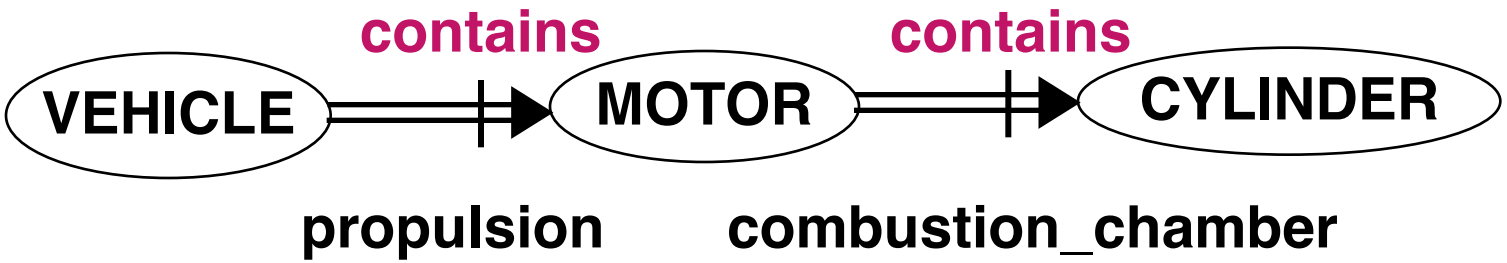
# Views Show Part of a Design



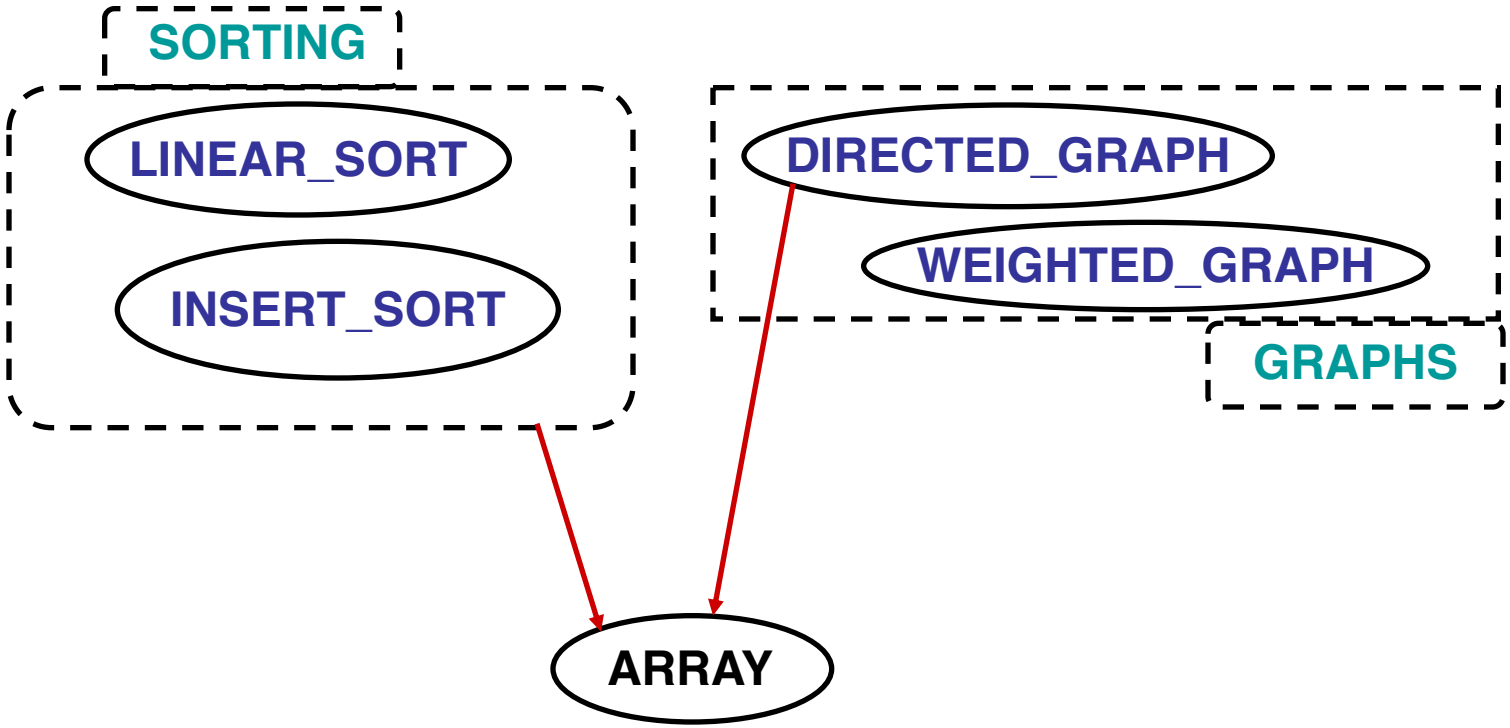

**PAPER** has other features not important for this view


uses
uses
PERSON
ADDRESS
CITY


CUSTOMER
shoppers : SET [ ... ]
preferred_mall
SHOPPING_CENTER


contains
contains
VEHICLE
MOTOR
CYLINDER
propulsion
combustion_chamber

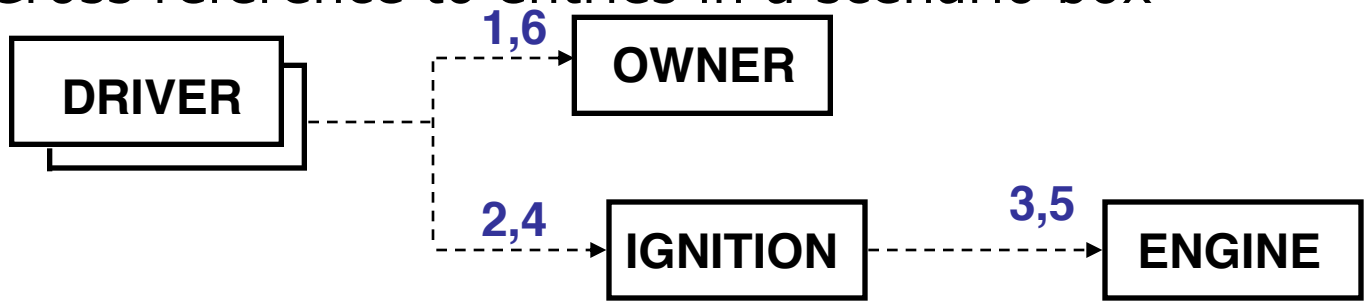
# Inheritance & Clusters




- All classes in SORTING inherit from ARRAY
- Only DIRECTED_GRAPH inherits from ARRAY

# Scenario with Object Communication

- Message links may be annotated with sequence numbers representing order of calls.
  - Cross reference to entries in a scenario box



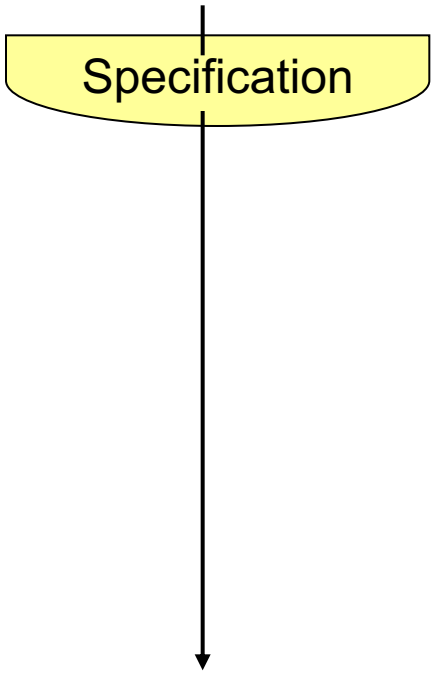

**Scenario:** Borrow car and go for a drive

1 Driver gets keys from owner
2 Driver turns ignition on
3 Engine starts
4 Driver removes key
5 Engine stops
6 Driver returns keys to owner

# The BON process

- B1 - **Delineate system borderline**: identify what the system will include and not include; define major subsystems, user metaphors, functionality, reused libraries.
- B2 - **List candidate classes**: produce first list of classes based on problem domain.
- B3 - **Select classes and group into clusters**: organize classes in logical groups, decide what classes will be deferred, persistent, externally interfaced etc.
- B4 - **Define classes**: expand the initial definition of classes to specify each of them in terms of queries, commands and constraints.
- B5 - **Sketch system behavior**: define charts for object creation, events and scenarios.
- B6 - **Define public features**: finalize class interfaces.
- B7 - **Refine system**.

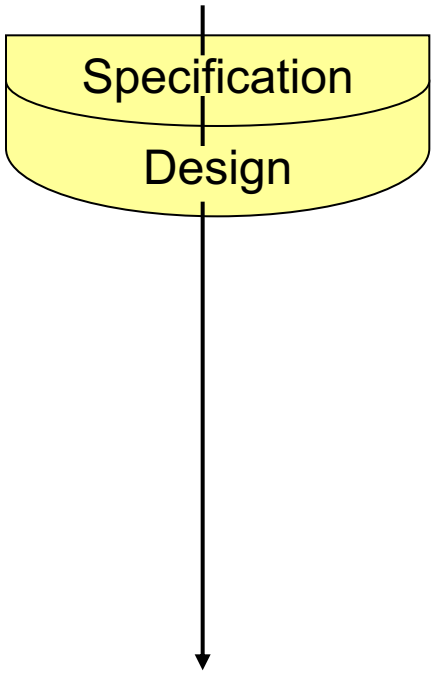
# Seamless development




TRANSACTION, PLANE, CUSTOMER, …

**Example classes**

# Seamless development

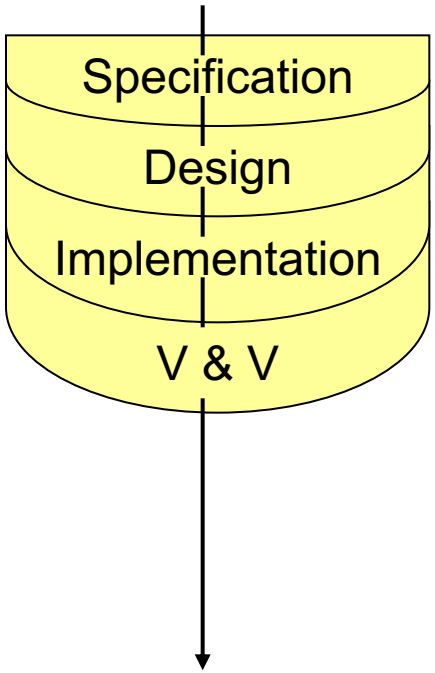



TRANSACTION, PLANE, CUSTOMER, …

STATE, USER_COMMAND, …

**Example classes**

# Seamless development




TRANSACTION, PLANE, CUSTOMER, …

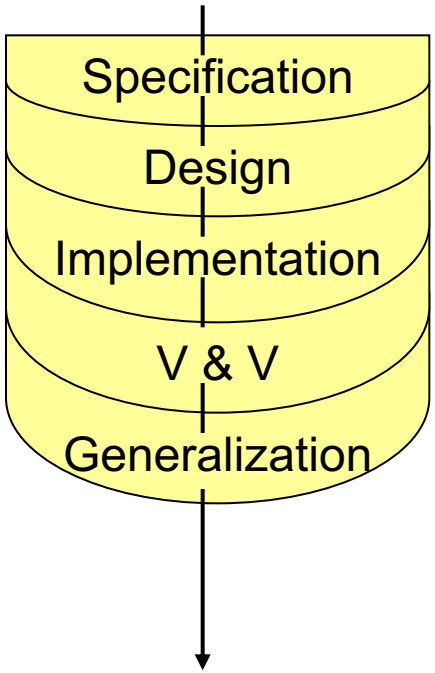
STATE, USER, …

HASH_TABLE, LINKED_LIST…

TEST_DRIVER, …

**Example classes**

# Seamless development




TRANSACTION, PLANE, CUSTOMER, …

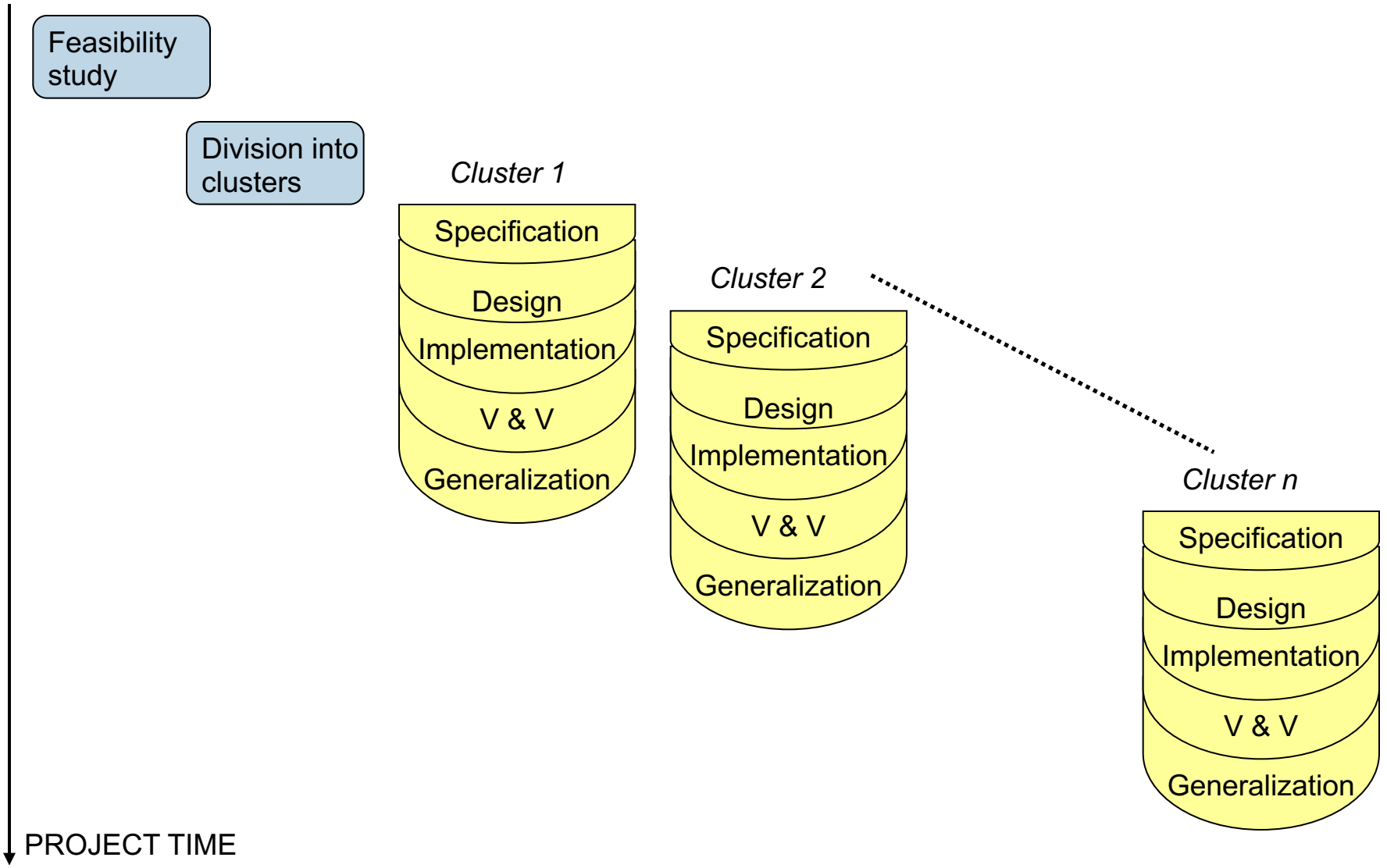
STATE, USER, …

HASH_TABLE, LINKED_LIST…

TEST_DRIVER, …

**Example classes**

# The cluster model



# Summary

- Quality First builds software
  - cluster by cluster
  - using a seamless
  - reversible process

# Cutter Vol. 24, #8

Table 1 — "Devops" in a Nutshell

| |
|---|
| Increased collaboration between operations and development |
| Reduced cycle times for operations activities (e.g., provisioning, deployment, change controls) |
| Extreme focus on automation in tools and processes |
| Fostering continuous improvement as both a means to the above and as a strategy to adapt to increasingly rapid changes happening in IT |

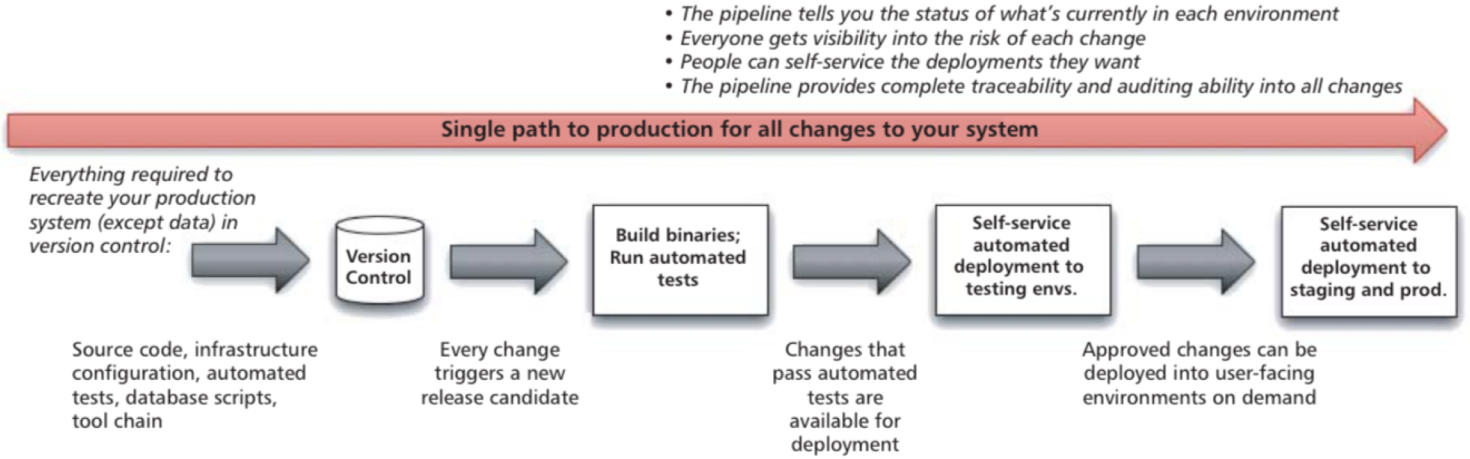



Figure 2 — The deployment pipeline.

Users

# Continuous Integration

- Maintain a code repository (Git, SVN etc)
- Automate the build. Automation of the build should include compile, testing, automating integration, deployment, production, documentation, website pages, measurement
- Make the build self-testing. Once the code is built, all tests should run to confirm that it behaves as the developers expect it to behave. Everyone commits to the baseline every day. By committing regularly, every committer can reduce the number of conflicting changes. Avoids: developer#1 checks in broken code and other developers merge new changes with it. Team can loses control of the system's working state, and suffer a loss in momentum when forced to revert changes from numerous developers to return to a functional state.
- Every commit (to baseline) should be built. The system should build commits to the current working version to verify that they integrate correctly.
- Keep the build fast. The build needs to complete rapidly, so that if there is a problem with integration, it is quickly identified.
- Make it easy to get the latest deliverables. Readily available to stakeholders and testers. All programmers should start the day by updating the project from the repository. That way, they will all stay up to date.
- Everyone can see the results of the latest build. It should be easy to find out whether the build breaks and, if so, who made the relevant change.
- Automate deployment