17-723: Designing Large-scale Software Systems

Design with Reuse

Tobias Dürschmid

This Lecture - Reuse

- What are **advantages** of reusing existing modules?
- What **challenges** might arise from reusing existing modules?
- How to **decide** whether to reuse a module?
- How to **reduce the risk of** negative consequences of reuse?

Case Studies: Many!

Carnegie Mellon University

Code Reuse vs. Design Reuse

Code Reuse

Including modules written for a different software in your own **code base**

This Lecture

Design Reuse

Abstracting the core idea of an implementation and **transferring** it to the design of a **similar problem.**

Lecture 6 - Generate

Carnegie Mellon University

Code Reuse vs. Design Reuse

Code Reuse

- Packages & Libraries
- Frameworks
- Software Product Lines

This Lecture

Design Reuse

- Design Patterns
- Tactics

Lecture 6 - Generate

Why Reuse?

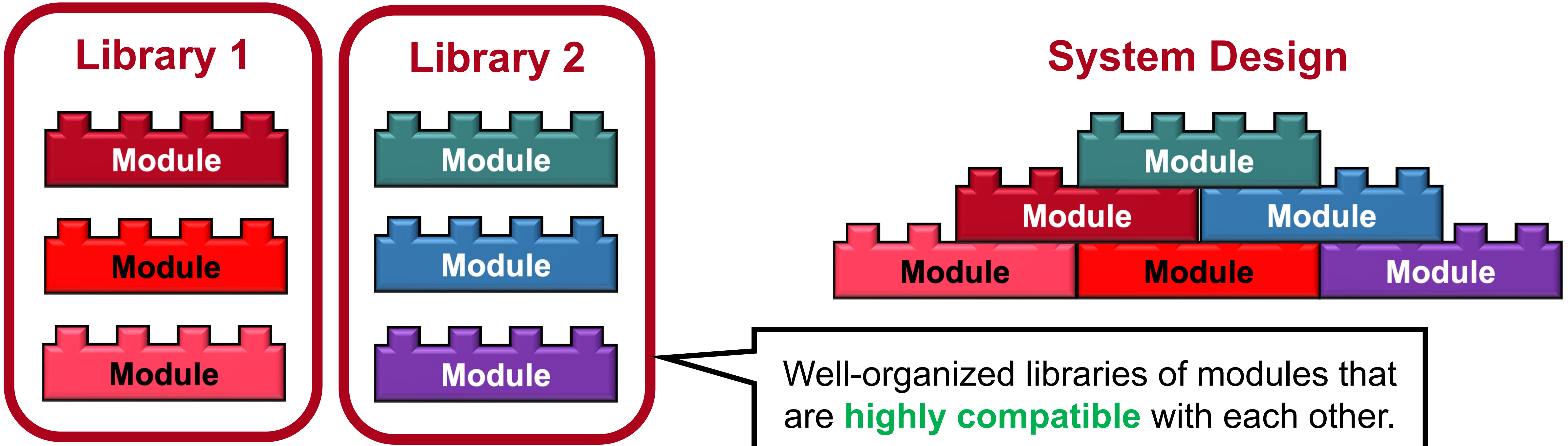
Higher Productivity / Faster Time to Market

Reusing software can **speed up software development,** because time for implementation and testing may be reduced.

Higher Software Quality / Fewer Defects

Reused software, which has been **tried and tested** in working systems, should be **more dependable** than new software, since most bugs have likely been found already by other users of the module.

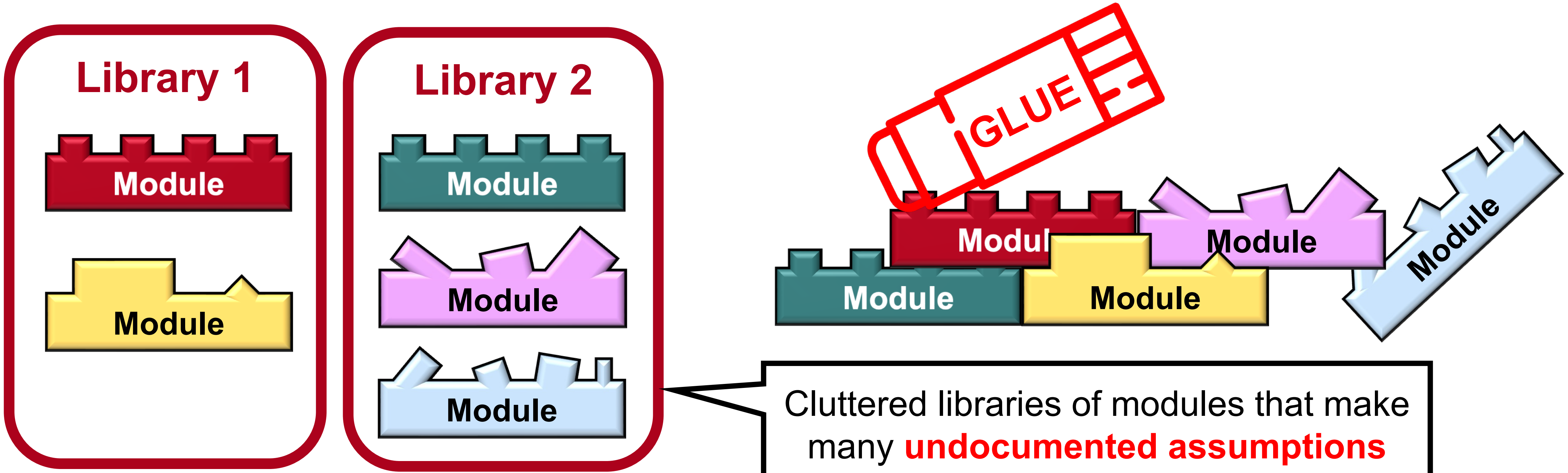
The Vision of Reuse: Creating New Software Mostly by **Composing Existing Building Blocks**



Well-organized libraries of modules that are **highly compatible** with each other.

Carnegie Mellon University

The Reality of Reuse: Modules are **Partially Incompatible** But Often **Still Glued Together**



Cluttered libraries of modules that make many **undocumented assumptions**

Carnegie Mellon University

Reuse must be Approached Differently Depending on its Source

Internal Reuse	External Reuse
Code was written by the **same developer, team, or organization** that is reusing it (e.g., product lines, component-based development process, …)	Code was written by a **third party**. (e.g., commercial off-the-shelf, open-source libraries, packages, frameworks)

Carnegie Mellon University

How to **Design with External Reuse?**

Designing Large-scale Software Systems - Design With Reuse

Carnegie Mellon University

Reusing 3rd Party Packages **Can Be Challenging**

Most common complaints by ROS Developers:

*"The package was for an **outdated** ROS distribution"*

*"I could not figure out **how to use** it"* (lack of documentation)

*"There was a **bug** that prevented the package from working properly"*

*"I did not succeed in **configuring** the package for **my use case**"*

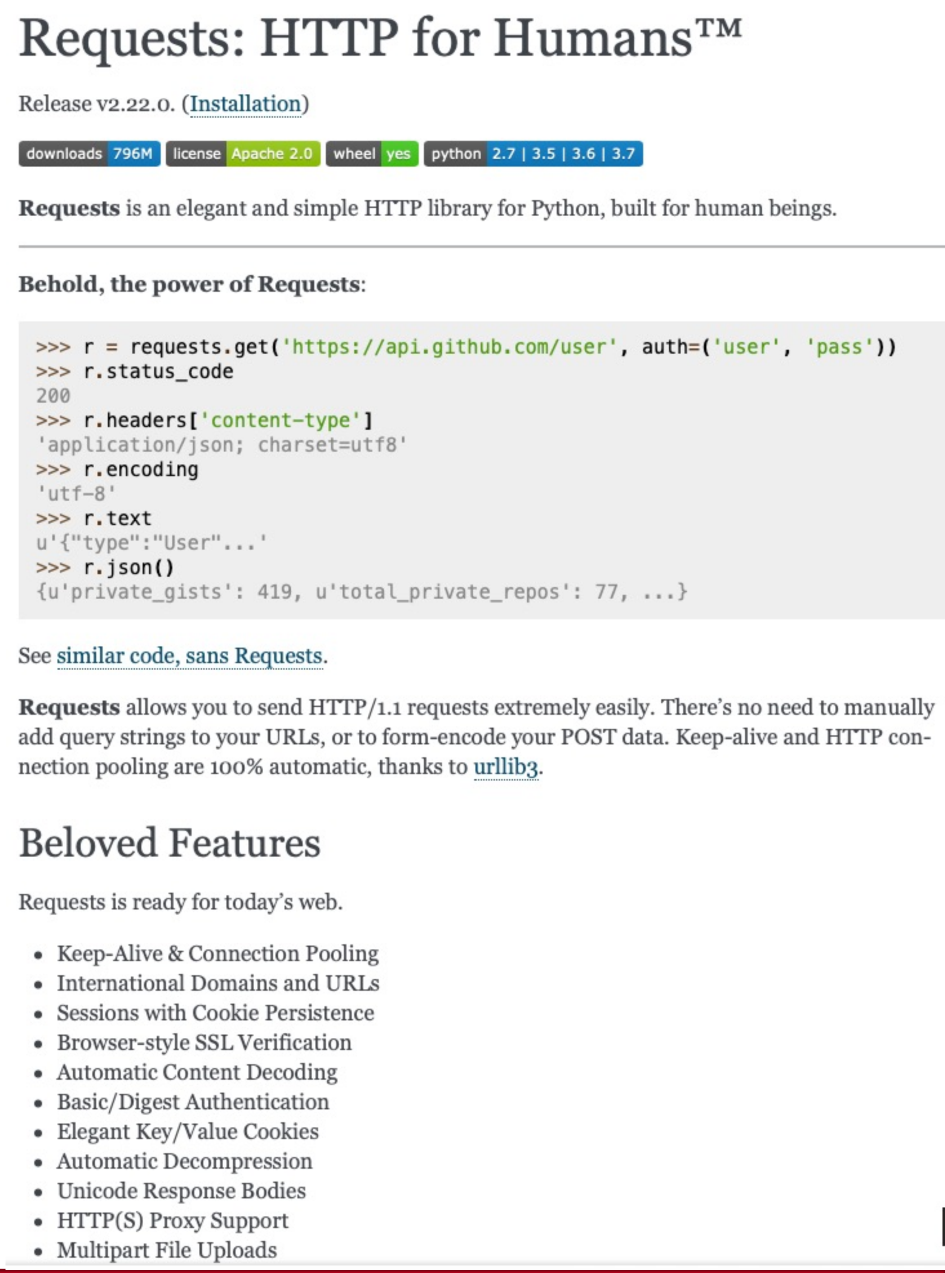
See "The Robot Operating System: Package reuse and community dynamics" (Estefo et al. 2019)

Carnegie Mellon University

The Python Ecosystem Is **Built on Reuse**

Most commonly needed functionality is already implemented in a reusable way

Importing & getting started with reusable modules is quite **easy**



Requests: HTTP for Humans™

Release v2.22.0. (Installation)

downloads 796M | license Apache 2.0 | wheel yes | python 2.7 | 3.5 | 3.6 | 3.7

Requests is an elegant and simple HTTP library for Python, built for human beings.

Behold, the power of Requests:

```
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type":"User"...'
>>> r.json()
{u'private_gists': 419, u'total_private_repos': 77, ...}
```

See similar code, sans Requests.

Requests allows you to send HTTP/1.1 requests extremely easily. There's no need to manually add query strings to your URLs, or to form-encode your POST data. Keep-alive and HTTP connection pooling are 100% automatic, thanks to urllib3.

Beloved Features

Requests is ready for today's web.

- Keep-Alive & Connection Pooling
- International Domains and URLs
- Sessions with Cookie Persistence
- Browser-style SSL Verification
- Automatic Content Decoding
- Basic/Digest Authentication
- Elegant Key/Value Cookies
- Automatic Decompression
- Unicode Response Bodies
- HTTP(S) Proxy Support
- Multipart File Uploads

Carnegie Mellon University

What can we learn from this example?

Example: Python Package Update Has **API-Breaking Change**

```
// in request package

httplib_response
= self._make_request(
    conn,
    method,
    url,
    timeout=timeout_obj,
    body=body,
    headers=headers,
    chunked=chunked,
)
```

Context: No source code changes
Python's **`docker`** package imports the **`request`** package and the **`urllib3`** package

Error Message: `docker.errors.DockerException:` **`Error while fetching server API version: request() got an unexpected keyword argument 'chunked'`**

Root Cause: `urllib3 2.0.0` just released today! And it changed its API to be **incompatible with `docker`**

Carnegie Mellon University

Design Principle: **Keep Versions of Your Dependencies Fixed**

- Most package managers allow you to **specify the versions of dependent packages** & install them in a virtual environment locally to the project
- E.g., Python: Use Pipenv & Pipfiles

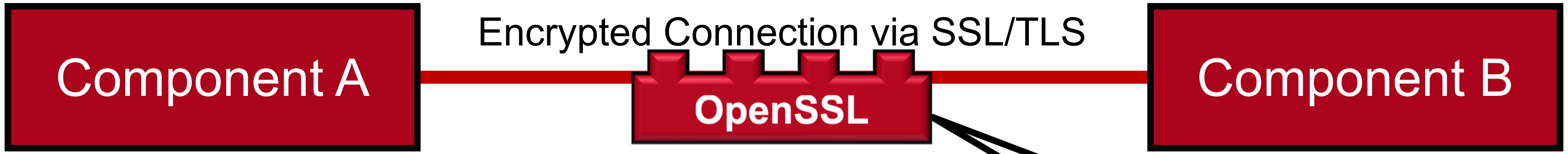
Example Pipfile

```
[packages]
flake8 = "==3.8.2”

[dev-packages]
flake8 = "==3.8.2"
pep8-naming = "==0.10.0"
mypy = "==0.910"
pytest = "==5.4.2"
tox = "==3.15.1"
coveralls = "==2.0.0"

[requires]
python_version = "3.9"
```

Carnegie Mellon University

What can we learn from this bug?

Heartbleed Bug in OpenSSL



Included **insecure implementation** of ***Heartbeat*** leading to a buffer over-read, leaking memory data

Introduced in February 2012

Discovered on 1 April 2014

Fixed version released on 7 April 2014

7 April 2014	17% of all secure web servers vulnerable
20 May 2014	1.5% of the most popular TLS-enabled websites still vulnerable
January 2017	180k internet-connected devices still vulnerable
July 2019	91k devices still vulnerable

Carnegie Mellon University

Design Principle: **Update Your Dependencies To Receive Bug Fixes**

- Defects in popular modules are usually fixed quickly
- Reusing **well-maintained** modules can improve your software quality
- Be aware of side effects of updates (see previous example)

Carnegie Mellon University

`left-pad` – A Simple and Highly Reused NPM Package

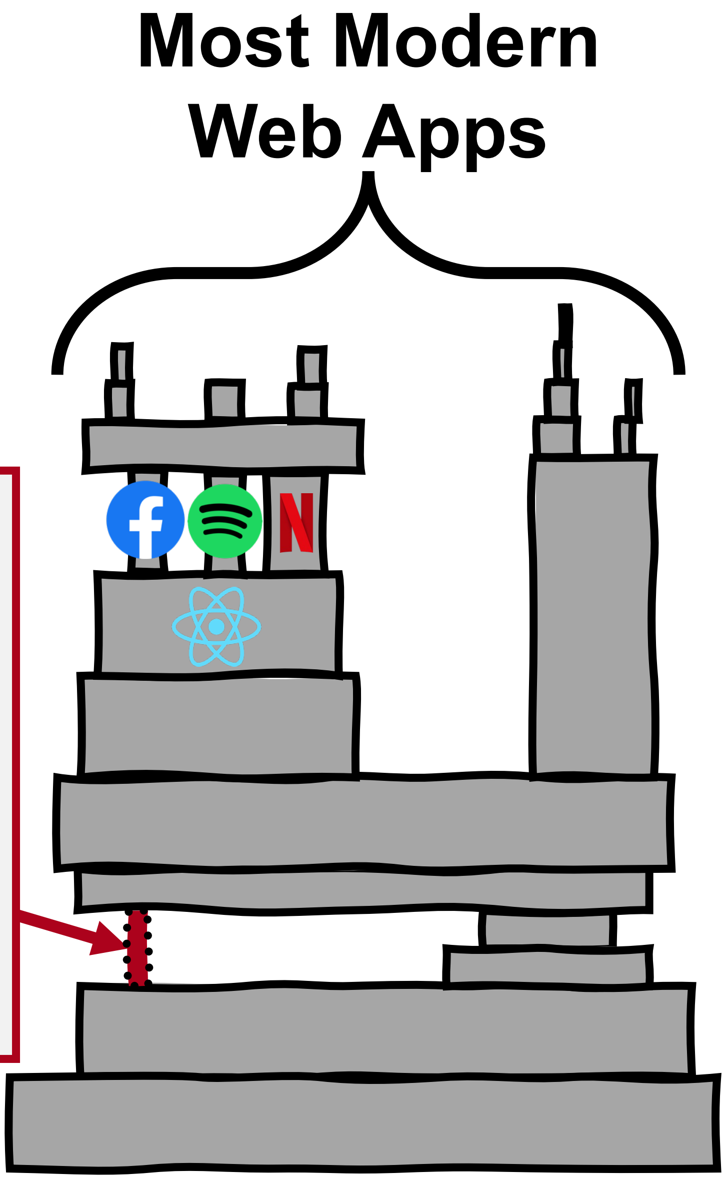
`left-pad` adds characters in front of a string for alignment in 11 lines of code.

Transitively, it is used in big popular packages (e.g., **`React`**, **`Bable`**), which are used by most modern web apps.

```
module.exports = leftpad;
function leftpad (str, len, ch) {
  str = String(str);
  var i = -1;
  if (!ch && ch !== 0) ch = ' ';
  len = len - str.length;
  while (++i < len) {
    str = ch + str;
  }
  return str;
}
```

left-pad

Stars on Github: 10
Time to Implement: ≈ 2 min
Weekly downloads: ≈ 1 million

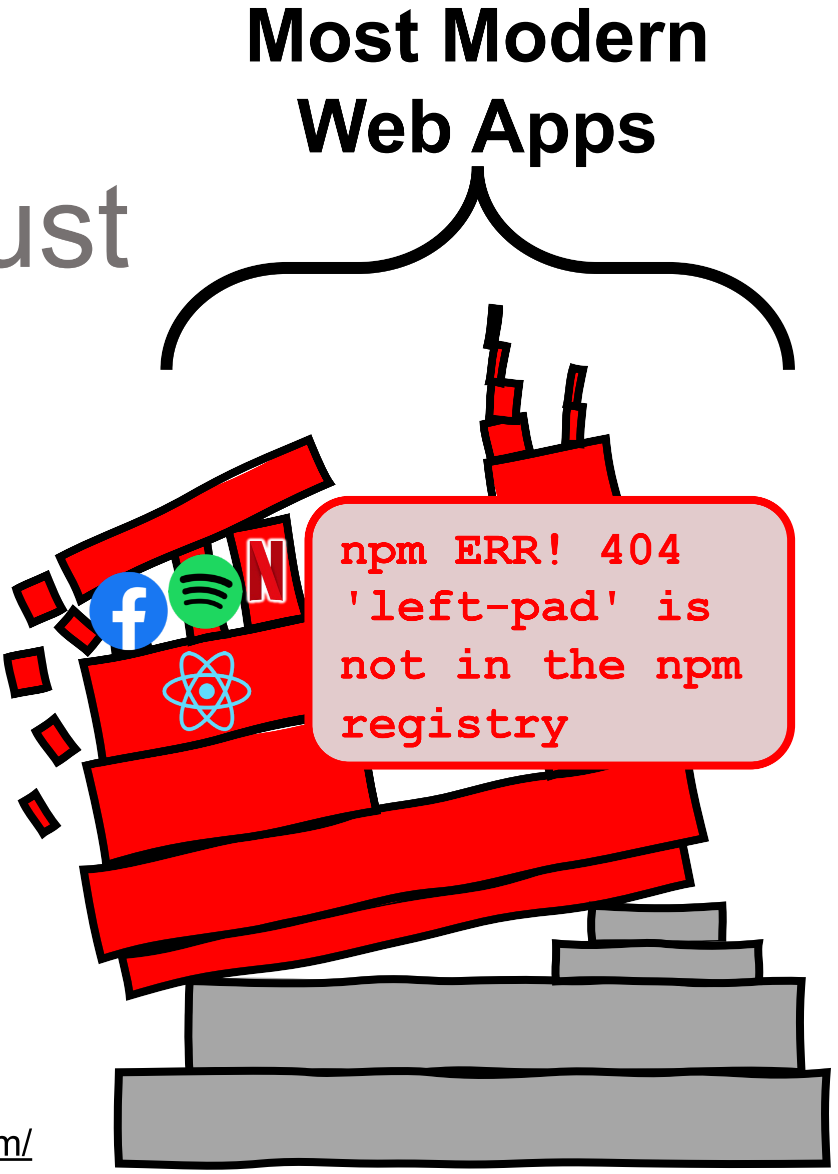


`left-pad` – How Reusing Just 11 Lines **Broke the Internet**

March 23, 2016: The author of `left-pad` decides to **un-publish** all his packages

Build processes for web apps across the internet **broke** due to the **missing package**

Many developers did not even know that they were **transitively relying** on `left-pad`



```
npm ERR! 404
'left-pad' is
not in the npm
registry
```

Read more here: https://www.davidhaney.io/npm-left-pad-have-we-forgotten-how-to-program/

Carnegie Mellon University

Learning from the `left-pad` story, **Describe Rules that Support Design With Reuse**

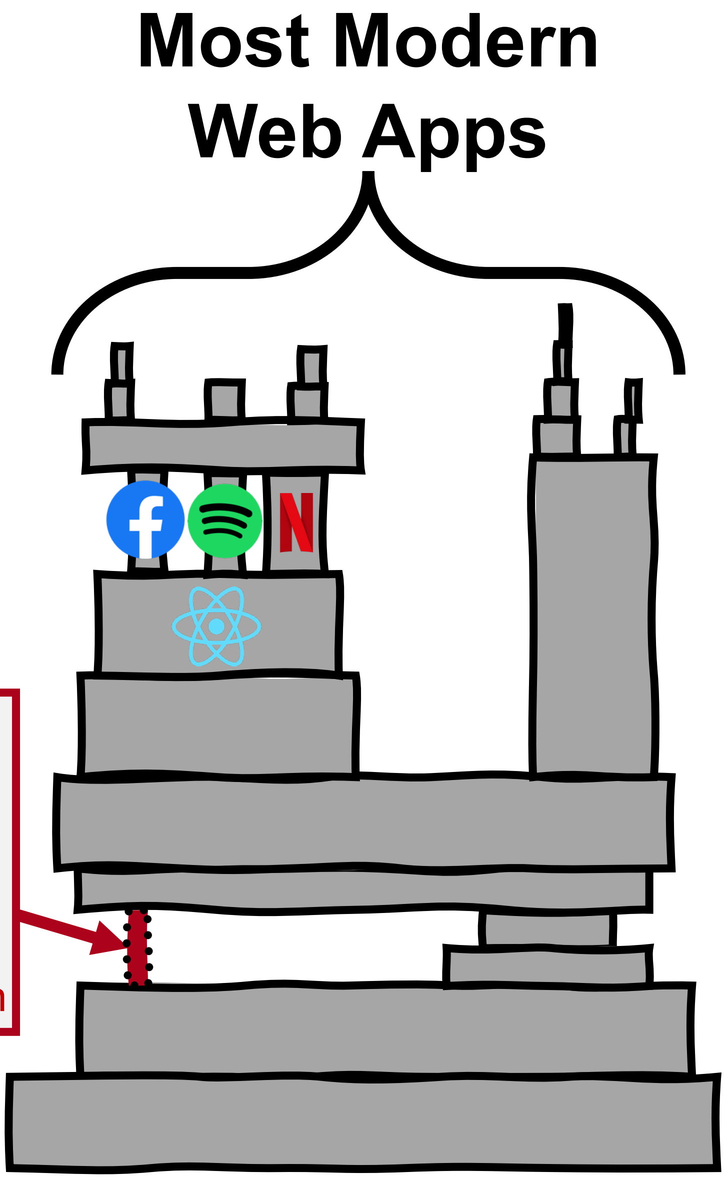
How should we decide **what to reuse**?

How can we **minimize the risk** of reuse?

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

isArray

Stars on Github: 129
Time to Implement: ≈ 1 min
Weekly downloads: ≈ 92 million



Carnegie Mellon University

Design Principle for Design With Reuse: **Strive for Fewer Package Dependencies**

- **Avoid reusing trivial code**, especially from unreliable sources
- **Carefully consider** adding new package dependencies
  - Every dependency can break, or **stop being supported**
  - Package dependencies can become a **security vulnerability**

    (e.g., `eslint-scope` malicious update)

    See https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes/

Modules with higher **Maintenance** Level & **Popularity** Are more **Viable** Reuse Candidates

- How actively does the development team **fix bugs** and update the module to support **new platforms**?
- **Popular** packages with many users are more likely to **resolve issues quickly** & have better **documentation**
- However, **fit to your context** is more important that popularity!

Carnegie Mellon University

Key-Takeaway from Previous Examples

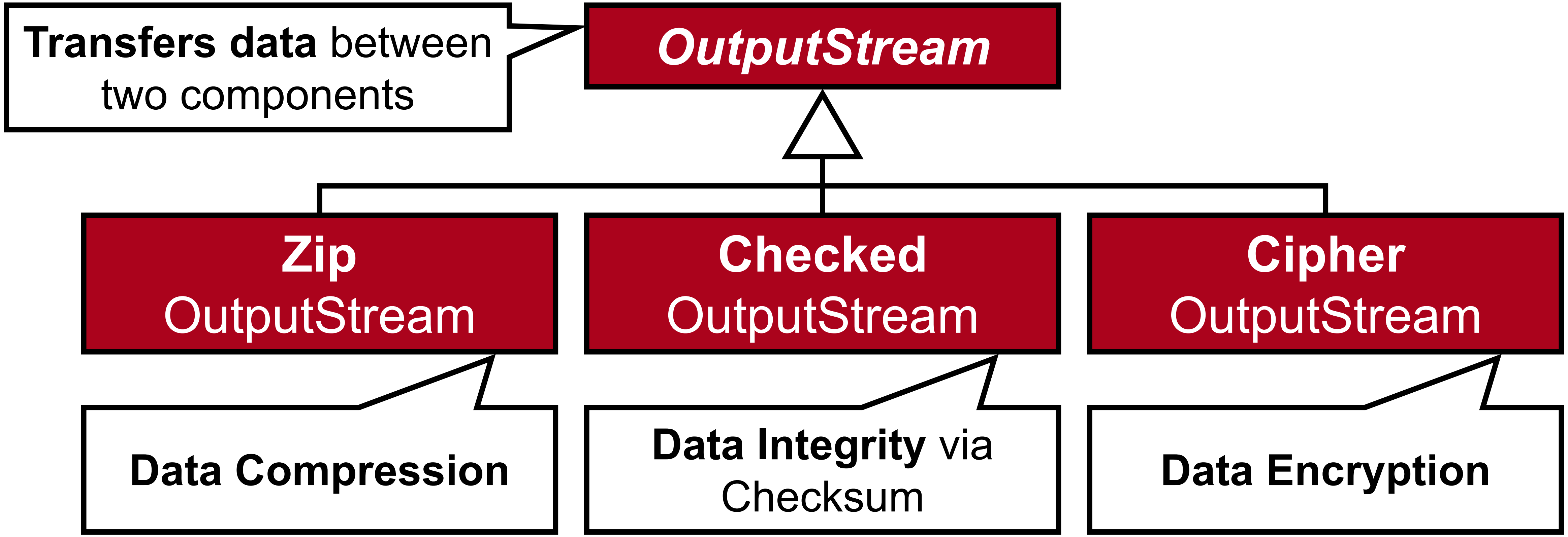
Lesson Learned: External Reuse Is **Not a One-Time Investment!**

- **Important updates** (e.g., fix security vulnerabilities) might come with **API-breaking changes** if you have skipped previous versions.
- Poorly maintained packages might require you to **abandon them later**
- Relying too much on reused code **limits changeability** once you need more than what the library offers.

Apply Design for Change

Carnegie Mellon University

Java Streams are Highly Reusable



Carnegie Mellon University

What makes Java Streams so Useful?

- Many Different Implementation of a very **Common Interface**
  - Supporting **Information Hiding** & **Changeability**
- Many domain-independent **Reuse Scenarios**
- Different Stream Implementations can be **Combined**!

Thanks to the *Decorator* Design Pattern

Carnegie Mellon University

Cost-Benefit Analysis for External Reuse

Effort to adapt the reusable module

Integration Effort (**Complexity,** Similarity of **Context**)

Updating Effort

Limiting **Changeability**

Effort saved reusing the module

Implementation Effort

Testing Effort

Benefit of **Update** Propagation

In-Class Exercise: Should you Reuse?

Context: Building an **appointment scheduling system**

Which of these packages are good reuse candidates? What are pros and cons of reusing them?

python-constraint
Provides a simple constraint satisfaction problem solver in Python to identify a scheduling solution for multiple users

icalendar
Generates, parses, and manipulates iCalendar data to send invitations to users

Carnegie Mellon University

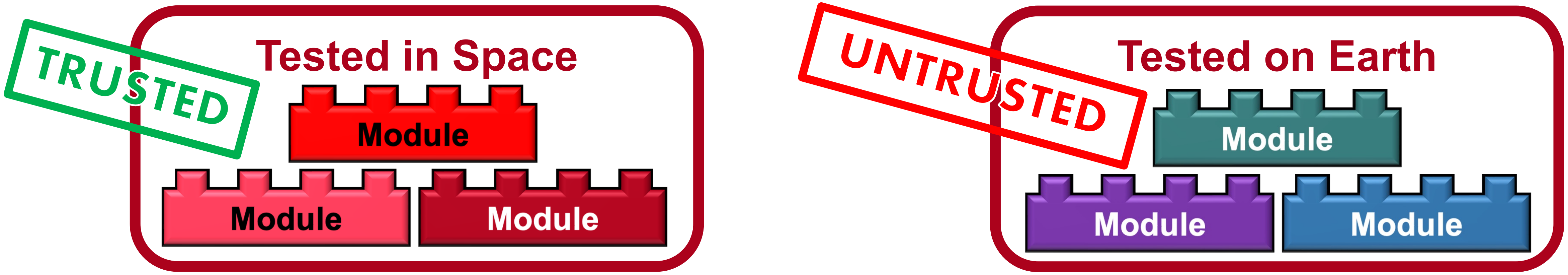
How to **Design with Internal Reuse?**

Designing Large-scale Software Systems - Design With Reuse

Carnegie Mellon University

NASA Heavily Relies on **Internal Reuse**

- **Problem**: Creating appropriate integration & system-level **tests** for space craft software is **difficult on Earth**
- **NASA's Solution**: Only trust software that has **worked in space**



What can we learn from this example?

Ariane 5 Failure

Can reach **higher velocities** than Ariane 4

Assumed **lower velocity**

Due to **Performance** Requirement

Ariane 4 Flight Control System

Worked Perfectly!

Inertial Reference System

Horizontal Velocity - **16 Bit Int**

Ariane 5 Flight Control System

Caused Self-Destruction

Inertial Reference System

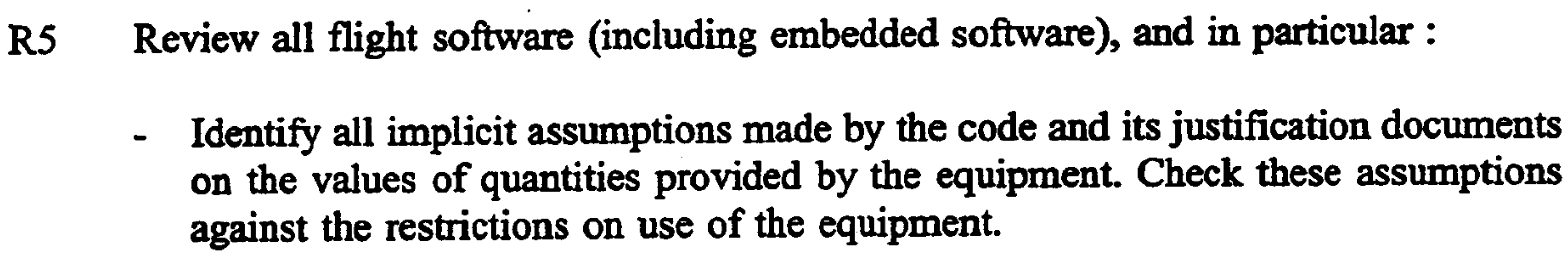
Horizontal Velocity - **16 Bit Int**

Overflow Error

See http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf

Based on This Example, Describe Rules that Support Design With Internal Reuse

Lesson Learned from Ariane 5 **Software that Worked in one Context Might Not Work in Another Context**



R5 Review all flight software (including embedded software), and in particular :

- Identify all implicit assumptions made by the code and its justification documents on the values of quantities provided by the equipment. Check these assumptions against the restrictions on use of the equipment.

See https://www.esa.int/Newsroom/Press_Releases/Ariane_501_-_Presentation_of_Inquiry_Board_report

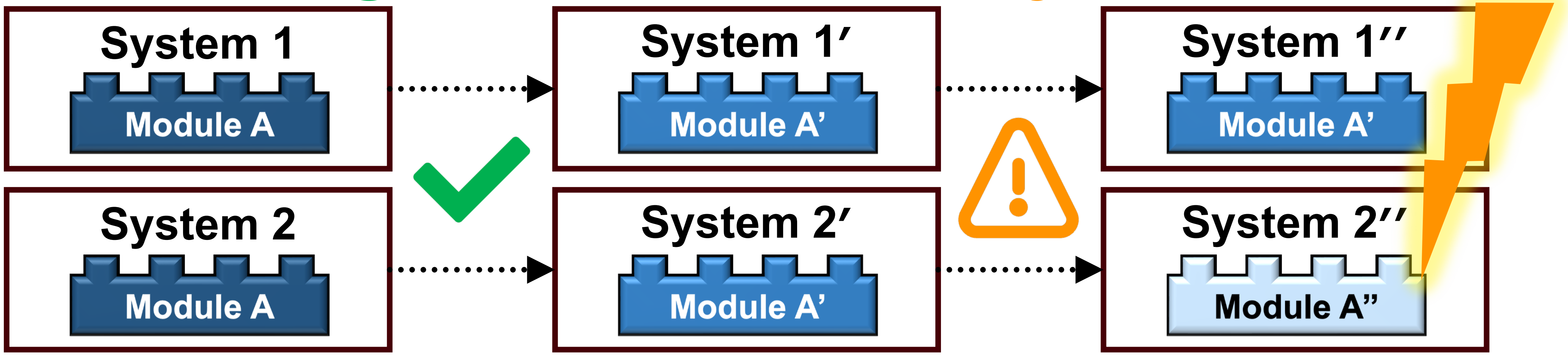
Carnegie Mellon University

Design Principle for Internal Reuse: **Identify Violated Assumptions**

- Check documentation and code to **identify assumptions** made by reuse candidate
- Check to make sure that reusable software was designed to operate reliably **under the conditions you want**
- Don’t **assume** the code of the reuse candidate is **correct, test it!**

Carnegie Mellon University

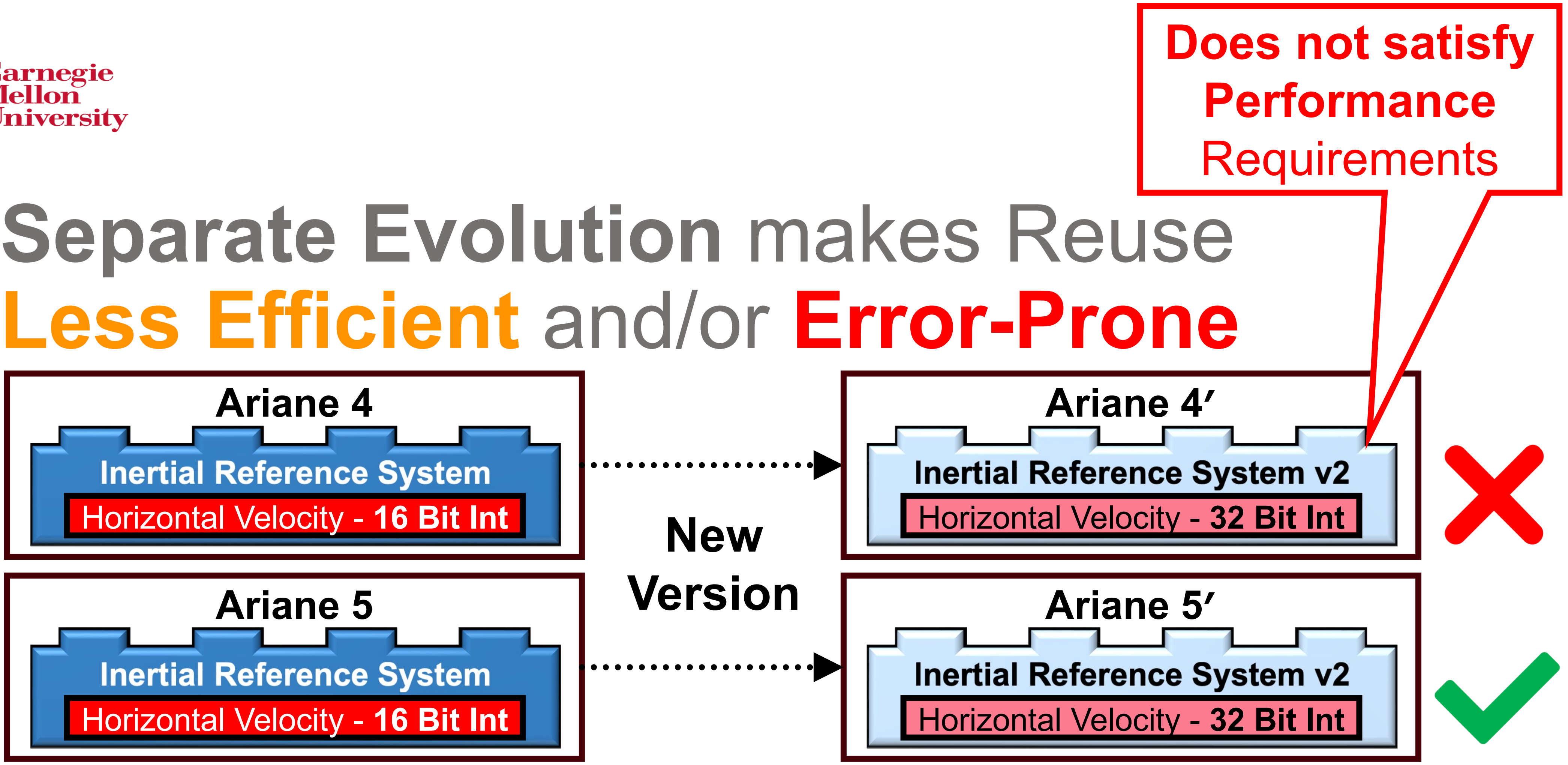
Consider Whether the Systems will **Evolve Together or Separately**



A change to a reusable module **impacts all systems** that reuse it.
Reuse is viable if the requirements of reusing systems change **together**.

Carnegie Mellon University

Separate Evolution makes Reuse Less Efficient and/or Error-Prone


Does not satisfy Performance Requirements
Ariane 4
Inertial Reference System
Horizontal Velocity - 16 Bit Int
Ariane 4′
Inertial Reference System v2
Horizontal Velocity - 32 Bit Int
New Version
Ariane 5
Inertial Reference System
Horizontal Velocity - 16 Bit Int
Ariane 5′
Inertial Reference System v2
Horizontal Velocity - 32 Bit Int

If systems evolve **separately**, consider **versioning the module** or "**clone & own**" (duplicating the code to allow independent evolution)

Carnegie Mellon University

Cost-Benefit Analysis for Internal Reuse

Effort to adapt the reusable module

Identification of Implicit **Assumptions**

Potential of **Separate Evolution**

Effort saved reusing the module

Implementation Effort

Testing Effort

Benefit of **Update** Propagation

Carnegie Mellon University

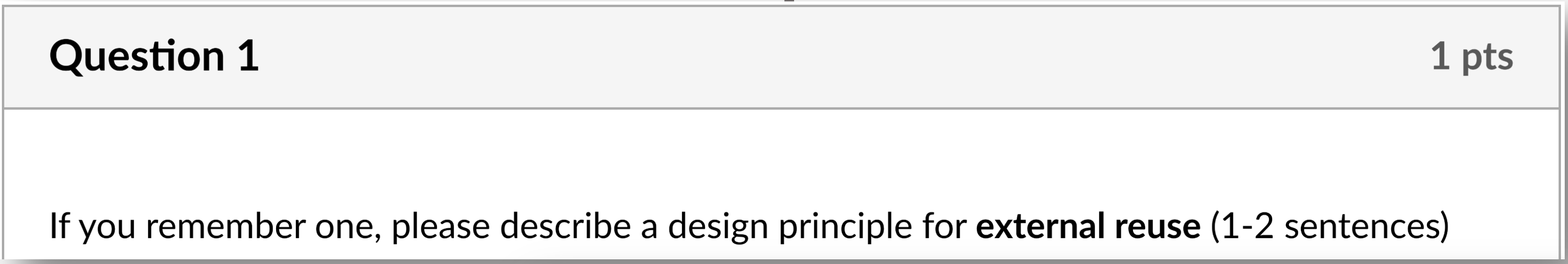
Scientific Evidence for **Real-World Benefits of Reuse**

	Internal Reuse	External Reuse
Higher Code Quality	✓	?
Higher Productivity	✓	?

See "What software reuse benefits have been transferred to the industry? A systematic mapping study" (Barros-Justo et al. 2017)

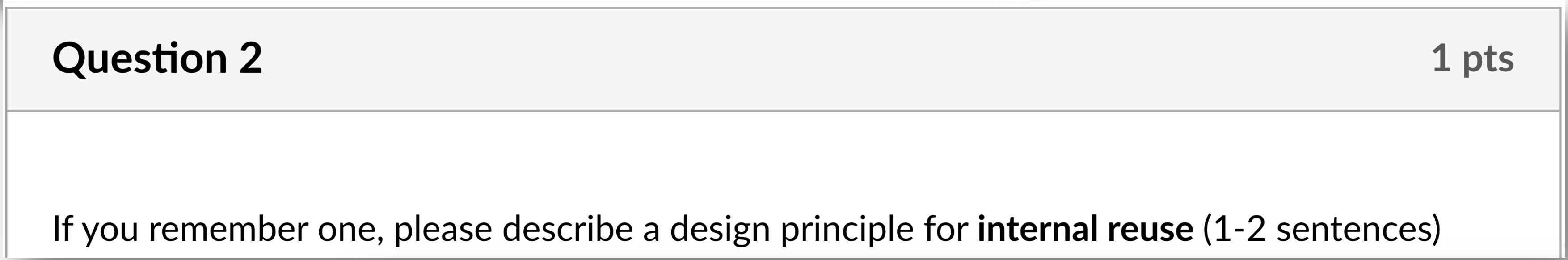
Carnegie Mellon University

Please Complete the Exit Ticket in Canvas!



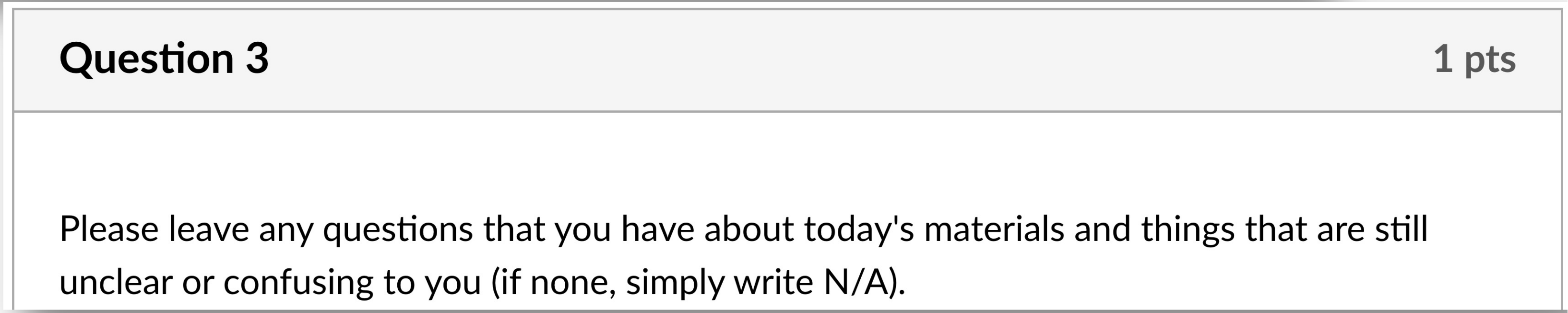
Question 1 1 pts

If you remember one, please describe a design principle for **external reuse** (1-2 sentences)



Question 2 1 pts

If you remember one, please describe a design principle for **internal reuse** (1-2 sentences)



Question 3 1 pts

Please leave any questions that you have about today's materials and things that are still unclear or confusing to you (if none, simply write N/A).