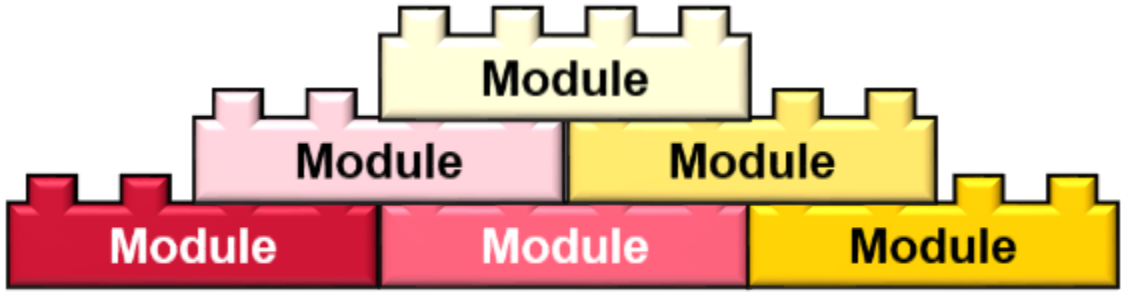
Carnegie Mellon University

# Design with Reuse

## 17-423/723 Designing Large-scale Software Systems

Tobias Dürschmid



Carnegie Mellon University

# This Lecture – Design with 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?

# **Why Reuse?** (Instead of Re-Implementing)

## Higher Productivity / Faster Time to Market

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

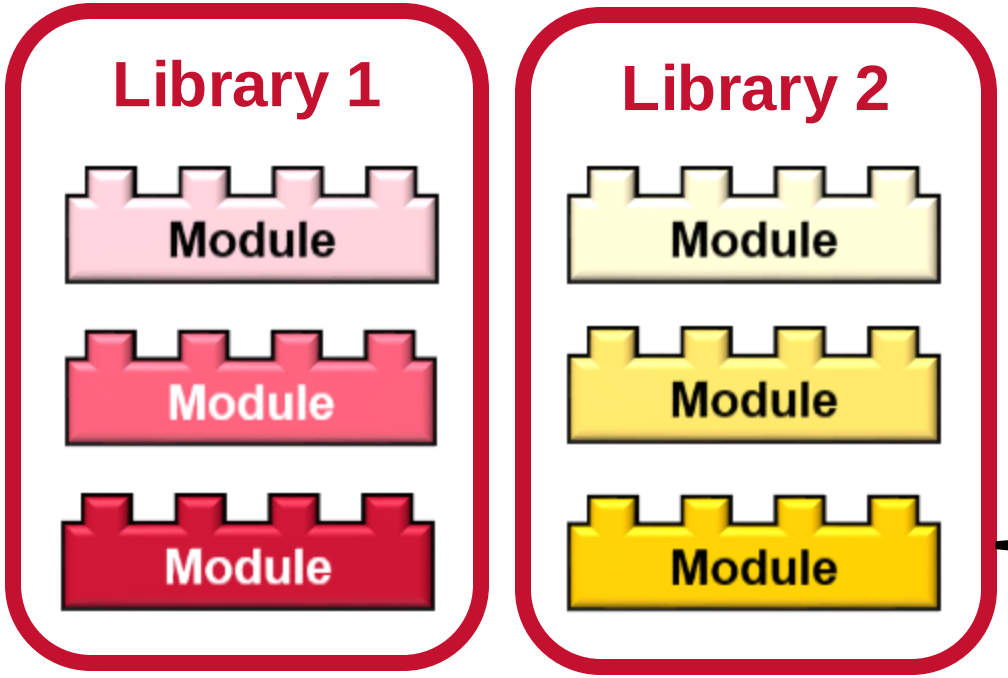
## Higher Software Quality / Fewer Defects

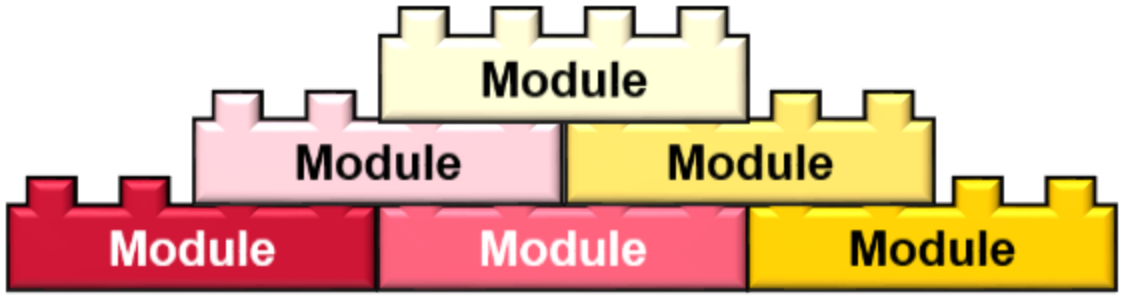
Reused software, which has been **tried and tested** in working systems, should be **more dependable** than new software, since most bugs have likely been found already by other users of the module.

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

Carnegie Mellon University

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



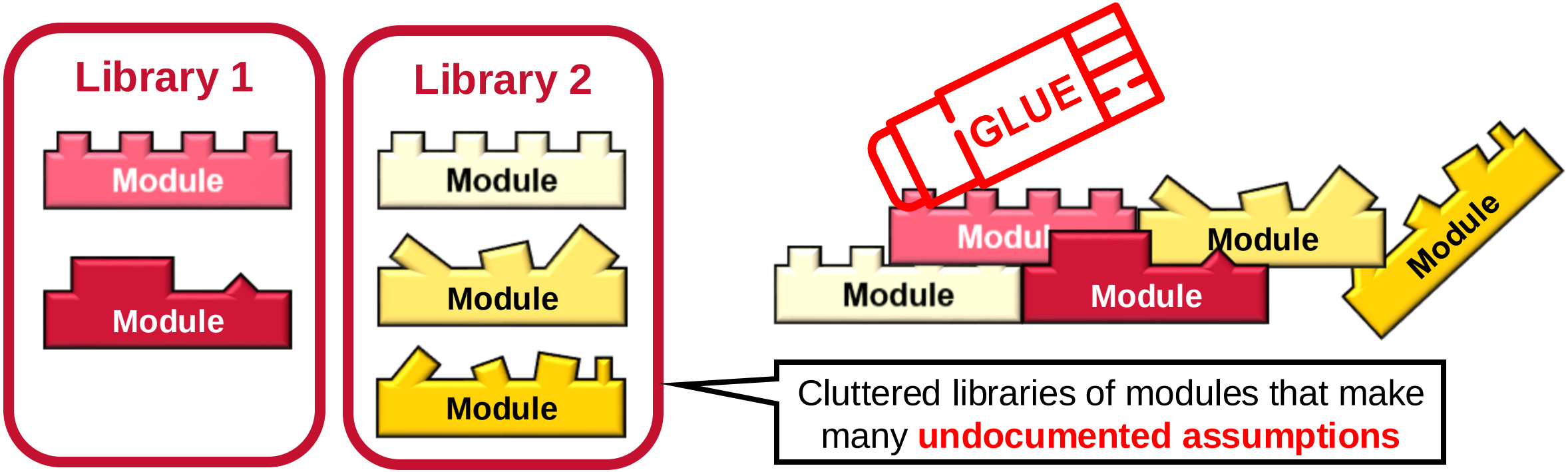




Well-organized libraries of modules that are **highly compatible** with each other.

Read more in: "Mass Produced Software Components" (Malcolm Douglas McIlroy 1968)

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**

Read more in: “Architectural Mismatch: Why Reuse Is (Still) So Hard” (David Garlan et al. 1995 and 2009)

Carnegie Mellon University

# Reuse must be Approached Differently Depending on its Source

## Internal Reuse

Code was written by the **same developer, team, or organization** that is reusing it (e.g., product lines, component-based development process, …)

## External Reuse

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

Tobias Dürschmid

Carnegie Mellon University

# The Python Ecosystem Is **Built on Reuse**

**Most commonly needed functionality** is already implemented in a reusable way

**Low Entry Barrier:** Importing & starting to use reusable modules is easy: `$ pip install requests`

```
>>> import requests
>>> response = requests.get("https://api.github.com")
>>> response.status_code
200
>>> response.json()
{'current_user_url': 'https://api.github.com/user', ...}
```

Carnegie Mellon University

**How can we prevent this from happening?**

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

**Context: No source code changes**

Python's **`docker`** package imports the **`request`** package and the **`urllib3`** package

```
// in request package

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

**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]
urllib3 = "<2.0.0"
docker = "==7.1.0"

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

[requires]
python_version = "3.9"
```

~~`$ pip install requests`~~ ✗ ~~`$ python <program>`~~

`$ pipenv install requests` ✓ `$ pipenv run <program>`

See more here: https://pipenv.pypa.io/

**Reusable Packages can introduce Security Vulnerabilities**

**What can we learn from this bug?**

# Heartbleed Bug in OpenSSL


Introduced in February 2012
Discovered on 1 April 2014
Fixed version released on 7 April 2014

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

| | |
|---|---|
| 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 |

# 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

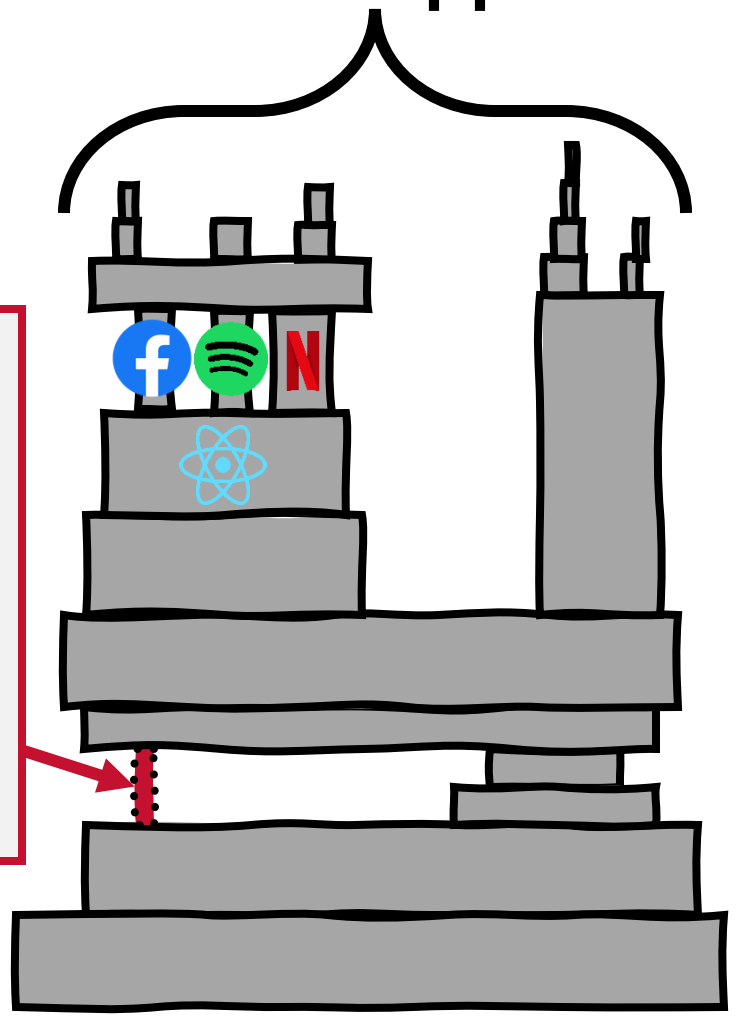
**`left-pad`** adds characters in front of a string for alignment with just 11 lines of code.

Transitively, it is used in big popular packages (e.g., **`React`**, **`Babel`**), 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
Weekly downloads: ≈ 1 million



Carnegie Mellon University

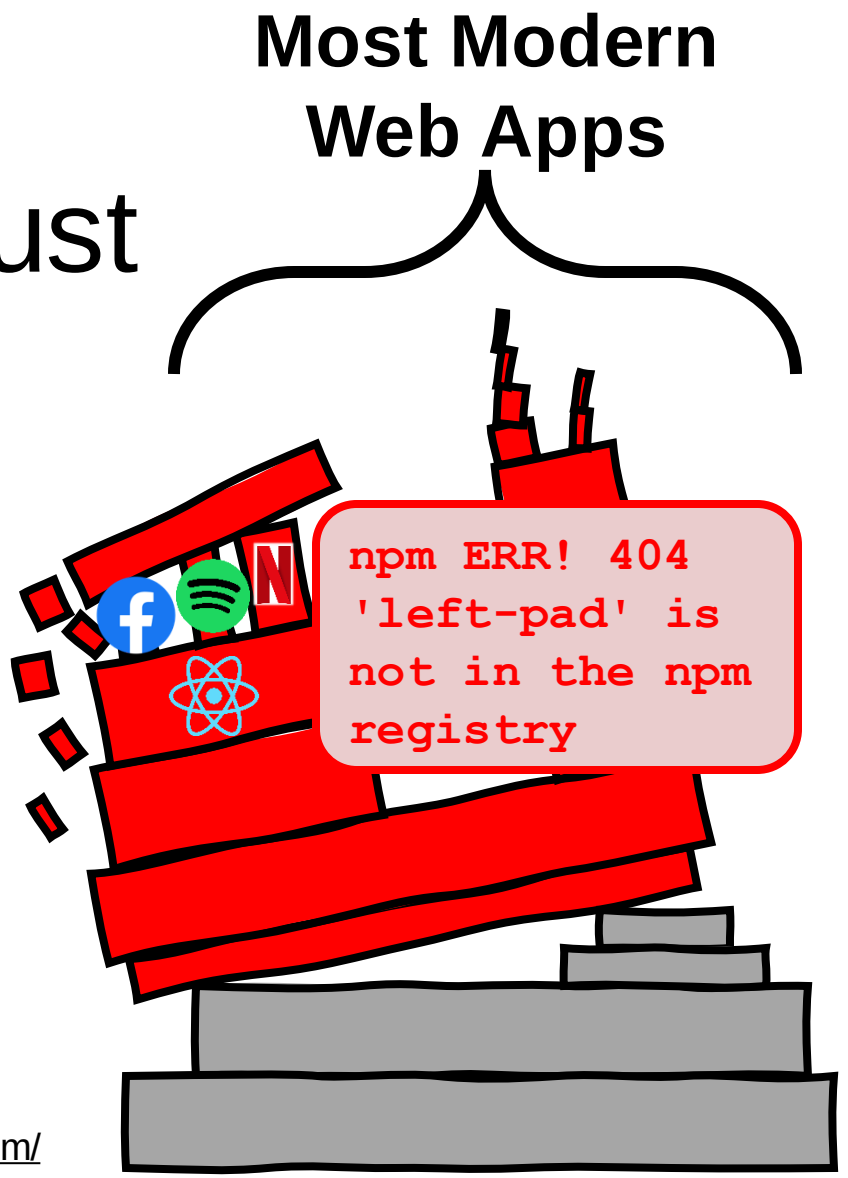
# `left-pad` – How Reusing Just 11 Lines **Broke the Internet**

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`**

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



Carnegie Mellon University

Small-Group Discussion

Talk to your neighbor!

# Learning from the `left-pad` story, **Describe Rules for Reusing Developers that Prevent Issues like that**

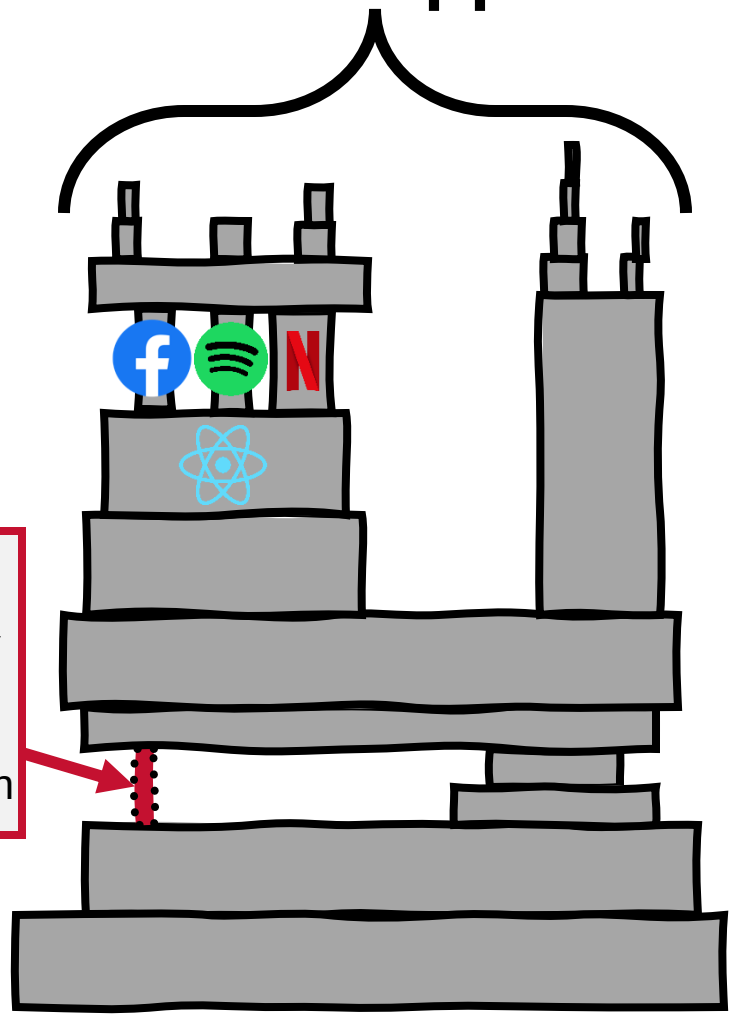
How should we decide **what to reuse**?

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

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

**isArray**

Stars on GitHub: 129
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 than popularity!

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.

Carnegie Mellon University

# **Cost-Benefit Analysis** for External Reuse

**Effort to adapt the reusable module**

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

**Finding** the Module

**Updating** Effort

Limiting **Changeability**

**Effort saved reusing the module**

**Implementation** Effort

**Testing** Effort

Benefit of **Update** Propagation

Read more here: Why reinventing the wheels? An empirical study on library reuse and re-implementation (Xu et al. 2019)

Carnegie Mellon University

# **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 (CSP) solver in Python to identify a scheduling solution for multiple users


Reusing a **large amount** of **hard-to-implement** functionality


**Limits changeability** (what if we want priority scheduling instead of global optimization?)

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


Reusing a **large amount** of **hard-to-implement** functionality


**High changeability** due to local change impact

Carnegie Mellon University

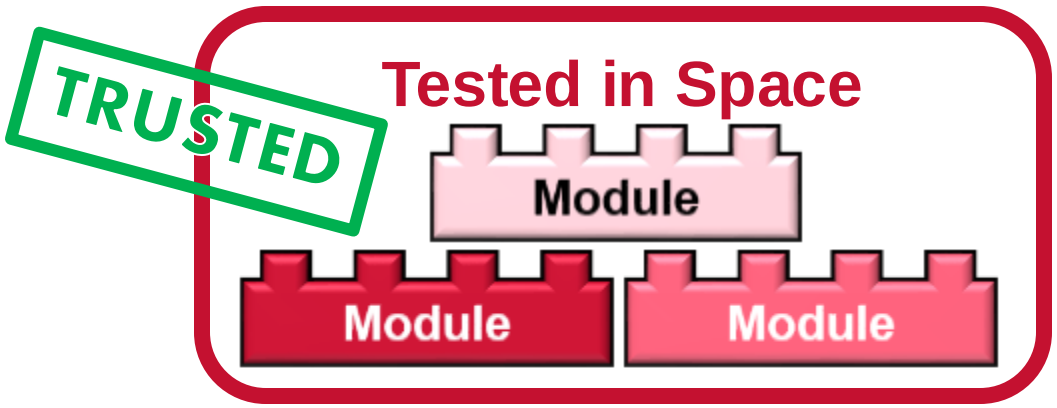
# How to Design with Internal Reuse?

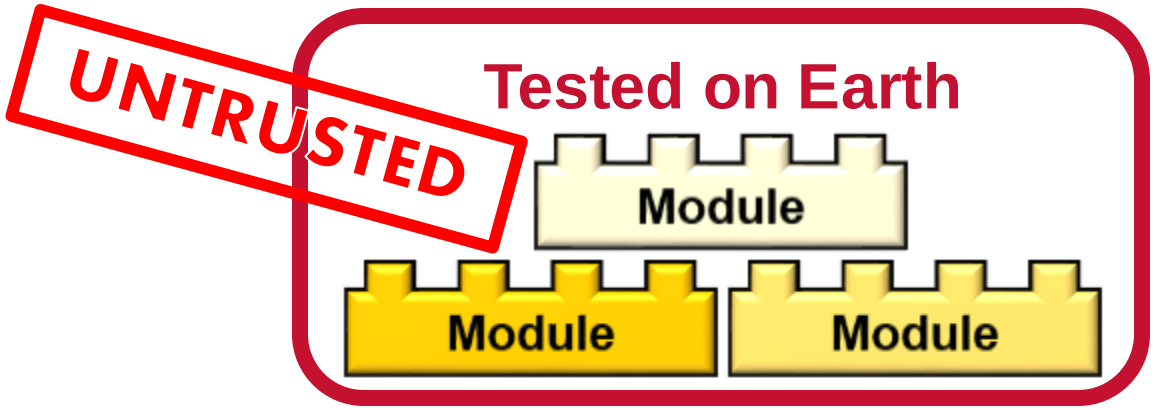
## Designing Large-scale Software Systems

Tobias Dürschmid

# 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**



**Describe Reuse Rules that Avoid Failures Like This**

Small-Group Discussion

# 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

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

# **Cost-Benefit Analysis** for Internal Reuse

**Effort to adapt the reusable module**

Identification of Implicit **Assumptions**

Effort to Create / Identify Reusable Modules

**Effort saved reusing the module**

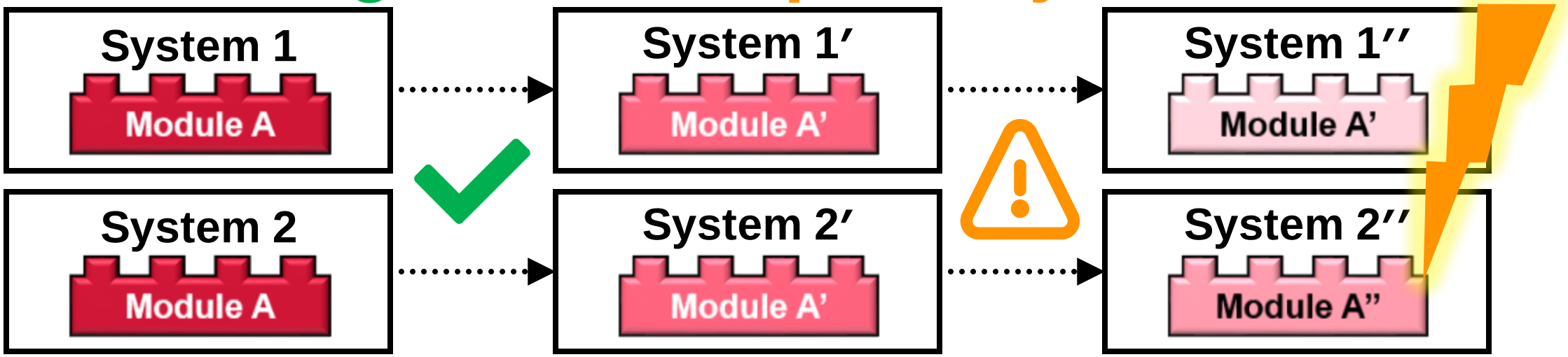
**Implementation** Effort

**Testing** Effort

Benefit of **Update** Propagation

Carnegie Mellon University

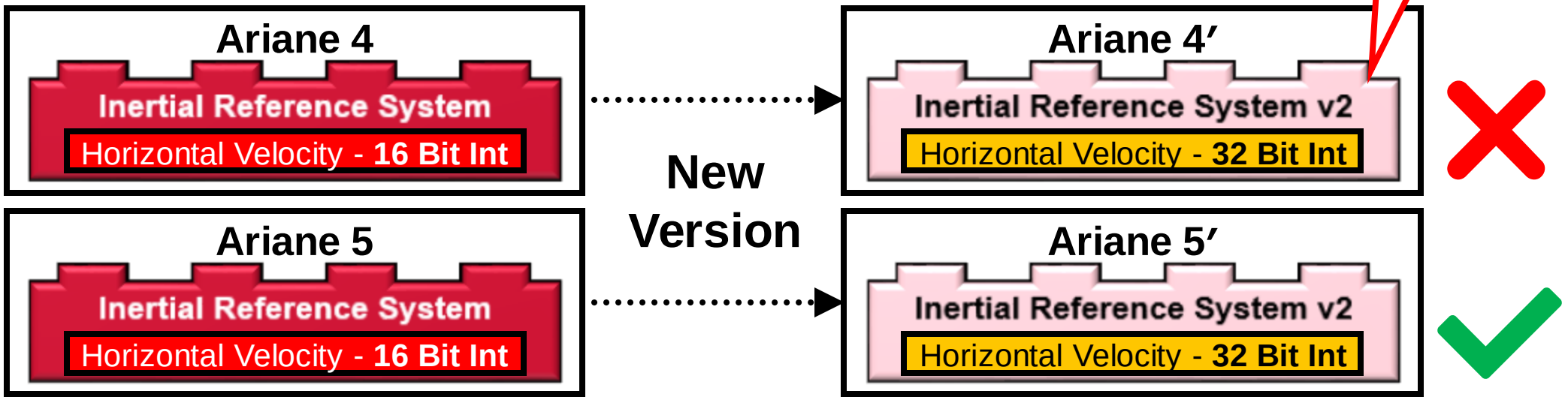
# Consider Whether the Systems will **Evolve Together or Separately**




A change to a reusable module **impacts all systems** that reuse it.

Reuse is viable if the requirements of reusing systems change **together**.

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




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

Carnegie Mellon University

# Summary / Exit Ticket

**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).

Credits: These slide use images from Flaticon.com (Creators: Freepik, surang, Eklip Studio)

# Summary

- Reuse can improve **development productivity** and **software quality**
- Strive for **Fewer Package Dependencies**
- External Reuse Is **Not a One-Time Investment**!
- **Identify Violated Assumptions**

Credits: These slide use images from Flaticon.com (Creators: Freepik, surang, Eklip Studio)