COMP2511

🎨 9.1 - Risk Engineering

In this lecture

- What is risk in Software Engineering?
- Mitigating risk
- Designing for Risk

The Flaw in the Plan

- Why do plans (designs) not go according to plan? What went wrong?
  - Flaws in the implementation / execution of the plan/design
  - Flaws in the design/plan itself
- We can't always plan for everything up front
- Design flaws are often hard to spot; Risk is invisible
- Can only tell through design smells / red flags
- Over time, we learn to become better at recognising warning signs and identifying flaws earlier on

- It's not what happened right before things went wrong that was the problem - it is what happened **every step along the way** that got us to that point

Design Debt, or Design Risk?

- ***Risk*** - the probability of a bad outcome occurring
- Design decisions come with a cost - "technical debt", the more technical debt, the more risk we accumulate
- Greater software complexity leads to more risk
- The design decisions and trade-offs we make are often the flaws in the plan - risks are inevitable
- How does this manifest itself?
  - Design problems often build in a "slow burn" fashion
  - Incidents, defects, bugs
  - Resistance to changes in software
  - These in turn present Business Risks

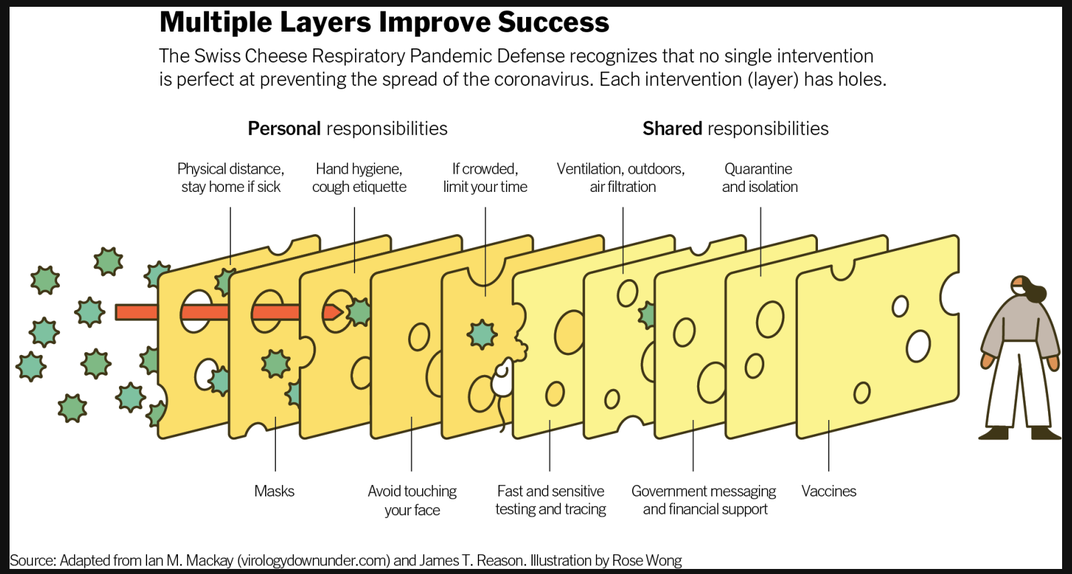
Mitigating Risk

- Risks are centred around events, e.g. software breaking.
- Risk is often assessed in terms of **probability** and impact
- Mitigations of **probability**
  - Preventative measures that lower the chance of a bad outcome occurring
  - E.g. Looking both ways before crossing the street
- Mitigations of **impact**
  - Reactive measures that decrease the negative outcome in the event that something bad does occur
  - E.g. Wearing a bike helmet
- This is often termed **Quality Assurance**

How do we design for risk?

Designing for Risk: Swiss Cheese Model

- James Reason - Major accidents and catastrophes reveal multiple, smaller failures that allow hazards to manifest as risks
- Each slice of cheese represents a barrier, each one of which can prevent a hazard from turning into consequences
- No single barrier is foolproof - each slice of cheese has "holes"
- When the holes all align, a risk event manifests as negative consequences

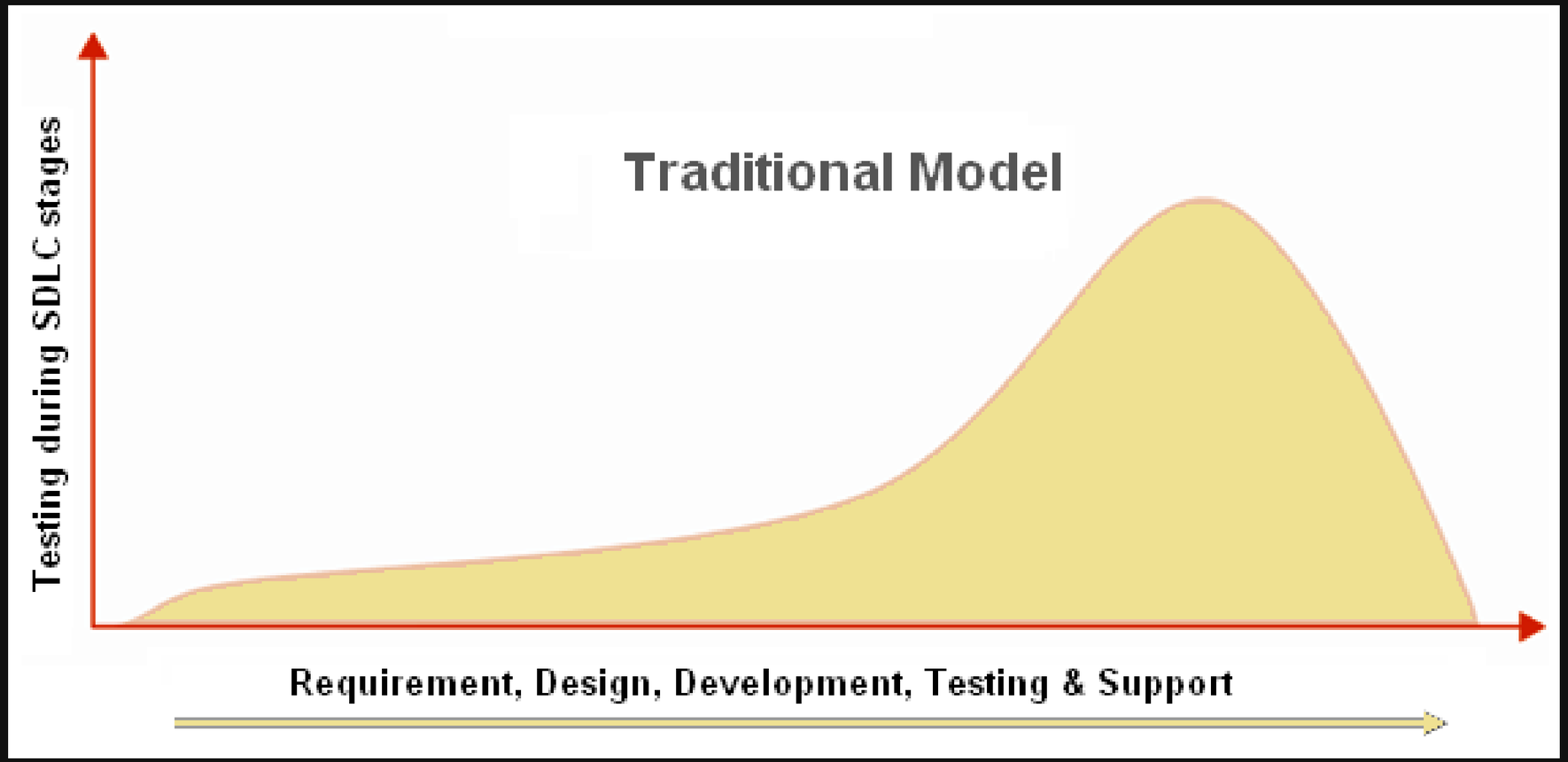


Designing for Risk: Swiss Cheese Model

- Taking a layered approach to Software Safety
- Testing at multiple levels:
  - Static verification
  - Unit and integration tests
  - Usability tests
  - Design and code reviews
  - CI pipelines
- Sometimes referred to as **containment barriers**
- A defensive approach; multiple checks and balances in place
- Probability is **multiplicative** (X AND Y AND Z = P(X) * P(Y) * P(Z))

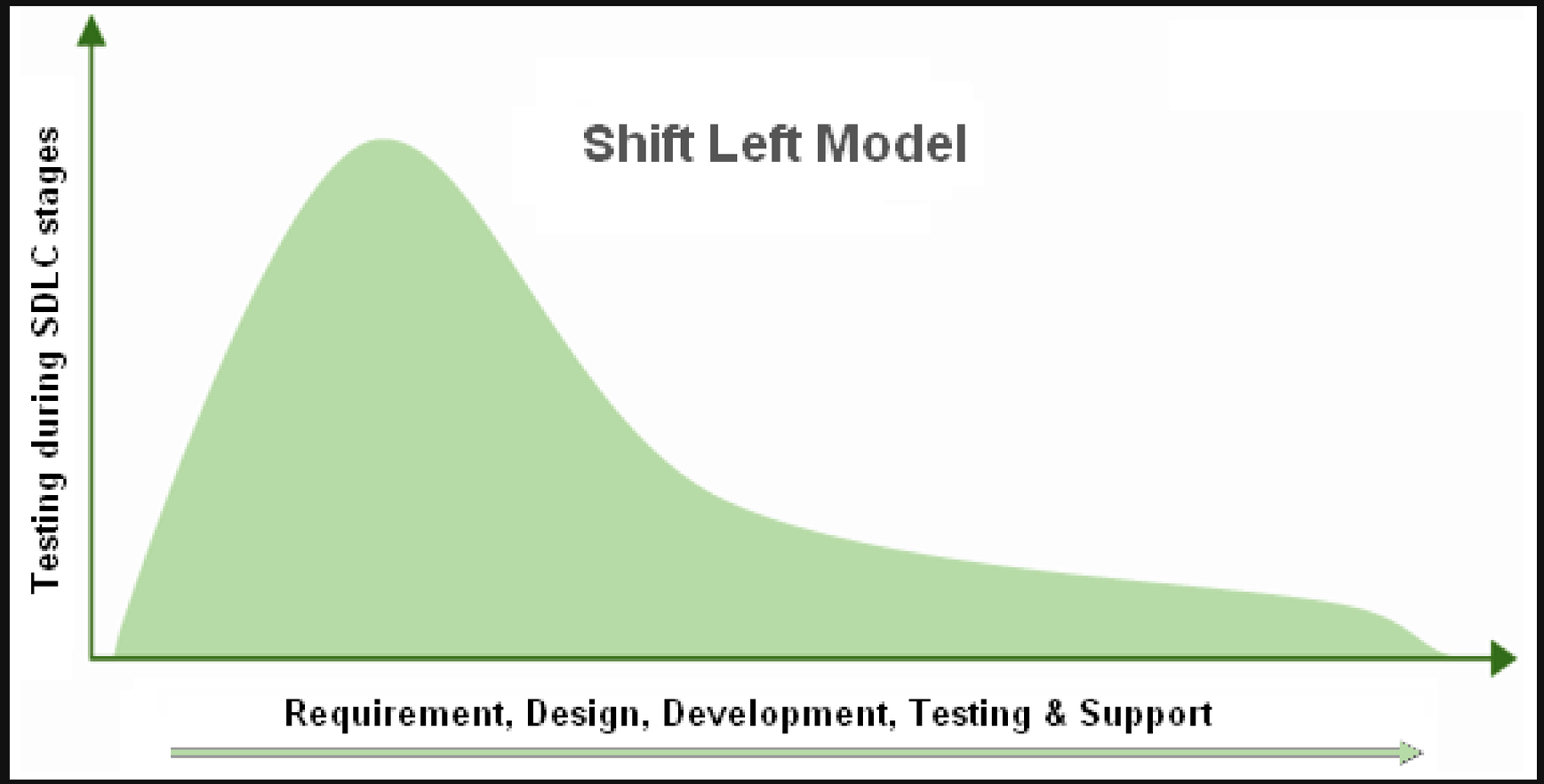
Designing for Risk: Shifting Left

A waterfall / big design up front approach to quality assurance.



Designing for Risk: Shifting Left

Shift Left: A practice intended to find and prevent problems early in the engineering process.



Designing for Risk: Shifting Left

- Shifting Left in principle: Moving risk forward in the software development timeline and designing systems and processes that are built for continuous testing
- What does shifting left involve in practice?
  - Automated testing over manual testing
  - Continuous Integration
  - Test-Driven Development

Shifting Left: An Example

- Let's take an example - a python script which runs on a remote server
- There is an error in the code, and the code fails when attempting to run a usability test

```
$ python3 -m svc.create_repo test
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/runpy.py", line 194, in _run_module_as_main
    return _run_code(code, main_globals, None,
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/runpy.py", line 87, in _run_code
    exec(code, run_globals)
  File "/Users/nicholaspatrikeos/Desktop/COMP2511-22T3/administration/svc/create_repo.py", line 11, in <module>
    PROJECT = gl.projects.get(f'{NAMESPACE}/{TERM}/STAFF/repos/{REPO}')
NameError: name 'REPO' is not defined
```

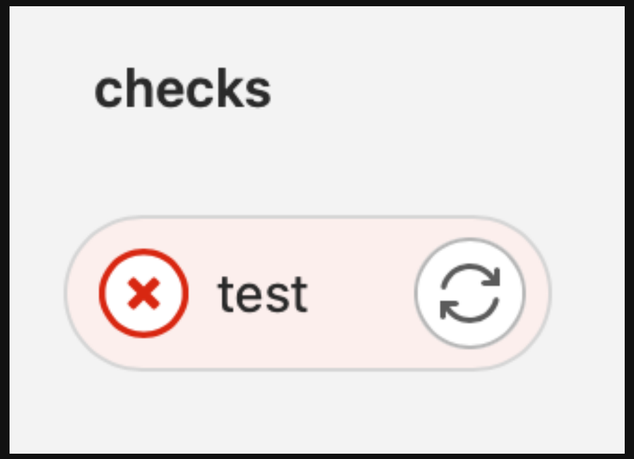
- How could we shift left here?

Shifting Left: Dynamic Verification + CI

- We can dynamically verify the correctness of the code and automatically run the tests in a pipeline:

```
$ pytest
============================= test session starts ==============================
platform darwin -- Python 3.8.8, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
rootdir: /Users/nicholaspatrikeos/Desktop/COMP2511-22T3/administration
plugins: hypothesis-6.1.1, xdist-2.2.1, timeout-1.4.2, forked-1.3.0
collected 1 item

create_repo_test.py F                                                    [100%]
```



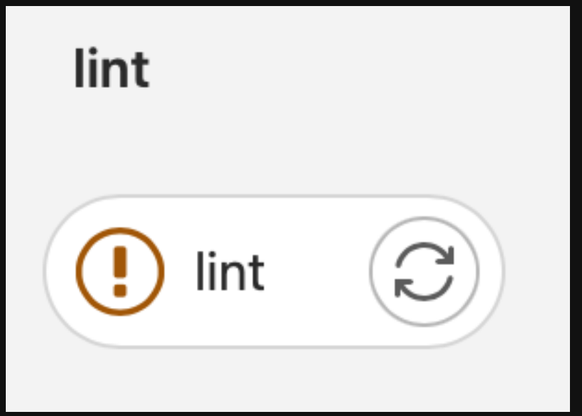
- Problem here - we are still having to run our tests in order to pick up a simple name error, this takes a long time to catch a small problem

Shifting Left: Static Verification + CI

- We can statically verify the correctness of our code, which is faster than running all the tests using a linter or a type checker:

```
$ pylint svc/*.py
************* Module svc.create_repo
svc/create_repo.py:11:64: E0602: Undefined variable 'REPO' (undefined-variable)
svc/create_repo.py:19:31: E0602: Undefined variable 'REPO' (undefined-variable)
svc/create_repo.py:27:24: E0602: Undefined variable 'REPO' (undefined-variable)

--------------------------------------------------------------------
Your code has been rated at 9.61/10 (previous run: 10.00/10, -0.39)
```



- Problem here - we are still having to push to the CI for our breaking changes to be contained. Can we enforce running them before?

Shifting Left: Local Configurations

- Pre-commit hooks and IDE tools can give us more friendly experiences that detect these problems earlier in the development loop, e.g.

- Ideally, static verification is "baked in" to our programming language rather than added on...

Shifting Left: Type Safety

- Types are **statically verifiable** - meaning that we can ensure correctness **earlier on in the development process**, shifting left
- In Java, code that doesn't adhere to the rules of the type system fails to compile - a significant containment barrier
- Extensions like mypy and TypeScript allow for an add-on of type checking
- Unlike Java however, type safety wasn't part of the Big Design Up Front for Python and JS
- Modern software design is favouring **statically typed languages** for these reasons

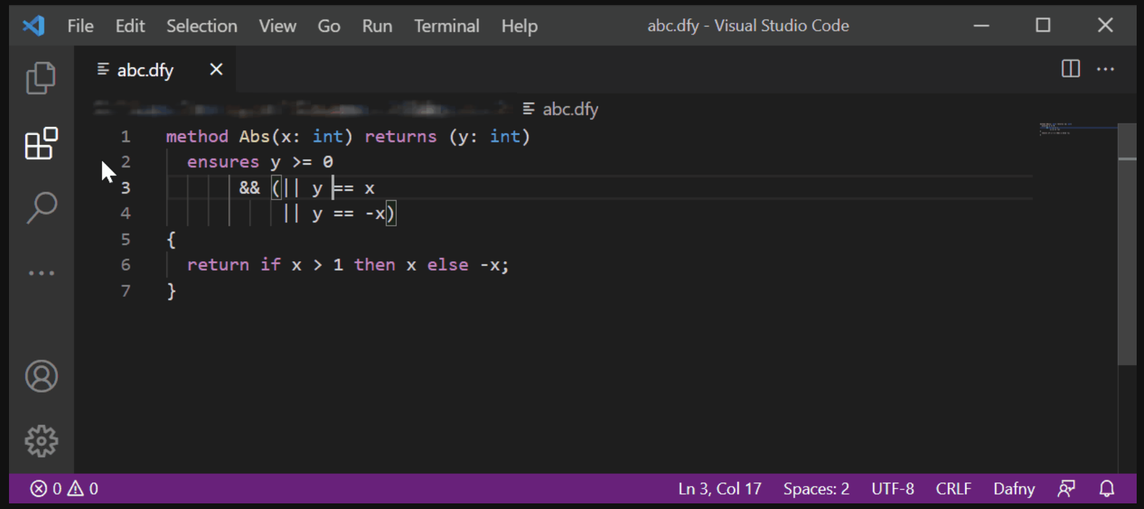
```
1 def my_function(message):
2     if message == 'hello':
3         return 1
4
5     return '0'
6
7 result = my_function('goodbye')
```

Shifting Left: Type Safety

- Features of type systems:
  - Ability to define custom types (typedefs)
  - Inheritance, Subtypes and Supertypes
  - Interfaces
  - Generics
  - Unit types
  - Enums
- Well-designed type systems allow us to verify more of our code statically

Shifting Left: More Static Verification & Design by Contract

- Some programming languages (e.g. Dafny) allow for more static verification than just type checking - they can prove or disprove code according to a **declarative contract** where preconditions, postconditions and invariants are specified
- Dafny makes use of a theorem prover which checks how well the implementation matches the specification (contractual correctness)



Summary

- Risk forms a large part of modern-day Software Engineering
- Designing for risk:
  - Considering risks in the design process;
  - Designing processes to accomodate for risk.
- Murphy's law: Anything that can go wrong, will.