Software Design Principles

Group Meeting

July 9, 2025

Henry Peterson

Why Care About Software Engineering?

- **Correctness**
- **Reproducibility**

Someone other than yourself should be able to run your code and change parts of it to understand what's happening

- **Maintenance**

How will you handle changes in the codebase when the requirements are updated?

SCIENTIFIC PUBLISHING

A Scientist's Nightmare: Software Problem Leads to Five Retractions

Dave's Software Levels:

1) Only you know how to run it
2) A close collaborator can run it
3) Any expert can run it

Sound familiar?

1) You write some code to do the thing
2) Now it has to do a slightly different thing
3) You copy-paste the old code into a new script/notebook and make some changes
4) Eventually, you have bunch of scattered code (and maybe the original even had a bug!)

Four Main Concepts for Today

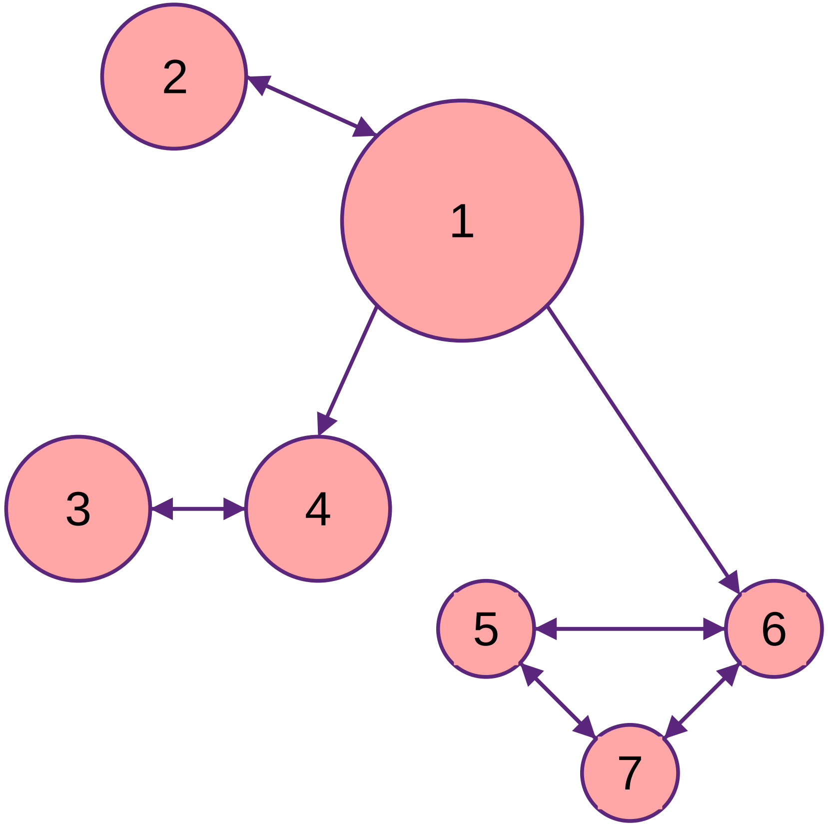
Minimize Complexity with:

1) Low Coupling
2) High Cohesion
3) High Testability
4) High Re-usability

In essence: build *small, sharp tools*.

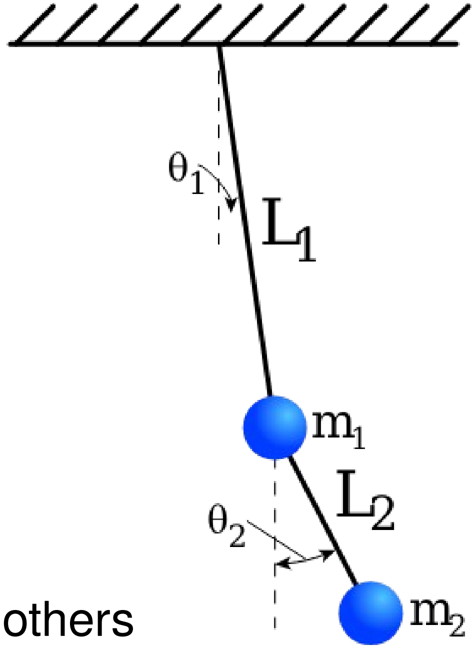
Software Components

- A **software component** can be many different things. It could be a
  - File
  - Function
  - Class
  - Program
  - …
- Our goal is to **manage the complexity** of our system by designing its components in a good way



Coupling of Components

- *How strongly coupled are your software components?*
- Examples of high coupling:
  - A script that relies on code/output from another script
  - A function that requires knowledge of another function
  - A class hierarchy
  - A cell in a Jupyter notebook
- For **highly coupled** software, changes in one component will affect others



Prefer the design that **minimizes coupling!**

Cohesion of Components

- *How broad/specific is the focus of your software component?*
- Examples of low cohesion:
  - A massive block of code inside a function/loop
  - One file (or notebook!) with tons of functions covering lots of functionality
- **Low cohesion** makes a component difficult to understand, hard to reuse, and dangerous to alter

Prefer the design that **maximizes cohesion!**

Coupling and Cohesion: Example

<see code in https://github.com/hgpeterson/Software_Example>

Testability

- How do you verify correctness of your science?
- How do you verify correctness of your code?
- What makes something easier to test?
  - Small pieces
  - Minimal setup
  - Well-defined input/output

Low coupling + high cohesion helps improve testability!

Testability: Example

```
def fit_power_law(m, ke, kmin=None, kmax=None, p0=None):
    """...
    ke_spec = np.mean(ke, axis=0)  # take mean over l

    def f(lk, lc, n):
        """...
        return lc + n * lk

    if kmin is None:
        kmin = m.k[0, 0, 0]
    if kmax is None:
        kmax = m.k[0, -1, 0]

    krange = np.where(np.logical_and(m.k[0, :, 0] >= kmin, m.k[0, :, 0] <= kmax))
    popt, pcov = curve_fit(
        f,
        np.log(m.k[0, krange, 0][0]),
        np.log(ke_spec[krange][:, 0]),
        p0=[np.log(p0[0]), p0[1]],
    )

    return np.exp(popt[0]), popt[1]
```

Reusability

- In science, it can be tempting to solve problems one script at a time. This can lead to:
  - Lots of repeated code
  - No version control
  - Time lost due to bugs in multiple locations
- In the long run, you'll save yourself time if you make your code **more reusable!**
  - Keeps you organized
  - Makes keeping track of versions easier
  - Bugs should only be in one place at a time ("Copying code copies bugs")
- Instead of the extreme "Don't repeat yourself" adage, try the **Rule of Three**:
  - "Do the thing. When you do it again, grimace but don't fuss. When you do it a third time, write a function!"

Re-usability: Examples

```python
def plot_2d_field(m, field, label, vmax=None, ilev=0, fname="2d_field.png"):
    """…
    if vmax is None:
        vmax = np.max(np.abs(field[:, :, ilev]))
    fig, ax = plt.subplots(1, figsize=(3.2, 2.4))
    im = ax.pcolormesh(
        m.x[0, :, ilev] / 1e3,
        m.y[:, 0, ilev] / 1e3,
        field[:, :, ilev],
        cmap="RdBu_r",
        vmin=-vmax,
        vmax=+vmax,
    )
    cb = plt.colorbar(im, ax=ax, label=label, shrink=0.6, pad=0.1)
    cb.ax.ticklabel_format(style="sci", useMathText=True, scilimits=(0, 0))
    ax.set_xlabel(r"$x$ (km)")
    ax.set_ylabel(r"$y$ (km)")
    ax.axis("equal")
    ax.set_xlim(0, m.a / 1e3)
    ax.set_ylim(0, m.a / 1e3)
    ax.set_xticks(np.array([0, m.a / 2, m.a]) / 1e3)
    ax.set_yticks(np.array([0, m.a / 2, m.a]) / 1e3)
    ax.spines["left"].set_visible(False)
    ax.spines["bottom"].set_visible(False)
    plt.savefig(fname)
    print(f"Saved '{fname}'")
    plt.close()
```

```python
def get_psi(m):
    """
    Calculate the streamfunction in spectral and physical space.

    The streamfunction is calculated by solving the linear system
    L * psi = q, where L is the linear operator defined by the model.

    Parameters
    ----------
    m : model object
        The model object containing grid information and the linear operator.

    Returns
    -------
    psi : ndarray
        The streamfunction in spectral space, shape (nl, nk, nz).
    psi_phys : ndarray
        The streamfunction in physical space, shape (n, n, nz).
    """
    psi = np.empty((m.l.size, m.k.size, m.nz), dtype=complex)
    for i in range(m.l.size):
        for j in range(m.k.size):
            psi[i, j, :] = np.linalg.solve(m.L[i, j, :, :], m.q[i, j, :])
    psi_phys = m.irfft2(psi, axes=(0, 1))
    return psi, psi_phys
```

Other Useful Tips

- Priorities: “Prepare to throw the first try away.”
- Priorities: “Premature optimization is the root of all evil.”
- Design: “Code is a liability, not an asset. What code does for you is an asset.”
- Design: “Everyone knows that debugging is twice as hard as writing a program in the first place. So if you’re as clever as you can be when you write it, how will you ever debug it?”
- Design, Documentation: “Any code of your own that you haven't looked at for six or more months might as well have been written by someone else.”
- Design: “Easy things should be easy, hard things should be doable.”
- Testing: “The longer a bug exists, the more expensive it is to fix.”

Extra Slides

Software Engineering != Programming

Added concepts: **Systematic, Disciplined, Maintainable**

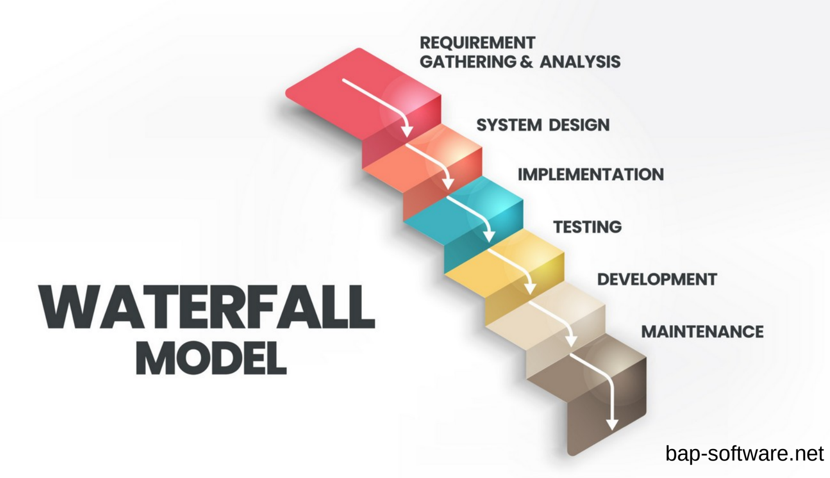
Phases of software development:

1) **Determine software requirements** *(What does it need to do?)*
2) **Design** *(Consider: user-interface, database schema, modularity, extensibility, security, safety, performance, etc.)*
3) **Implement**
4) **Test** *(Verification: Does the code do what I thought it should? Validation: Does the code do what the user thought it should?)*
5) **Document**
6) **Maintain**

Waterfall

- **Idea:** Try to complete each phase before the next phase begins
- **Issues:**
  - Impossible to know how long each step will take without experience
  - User doesn't see product until testing/release

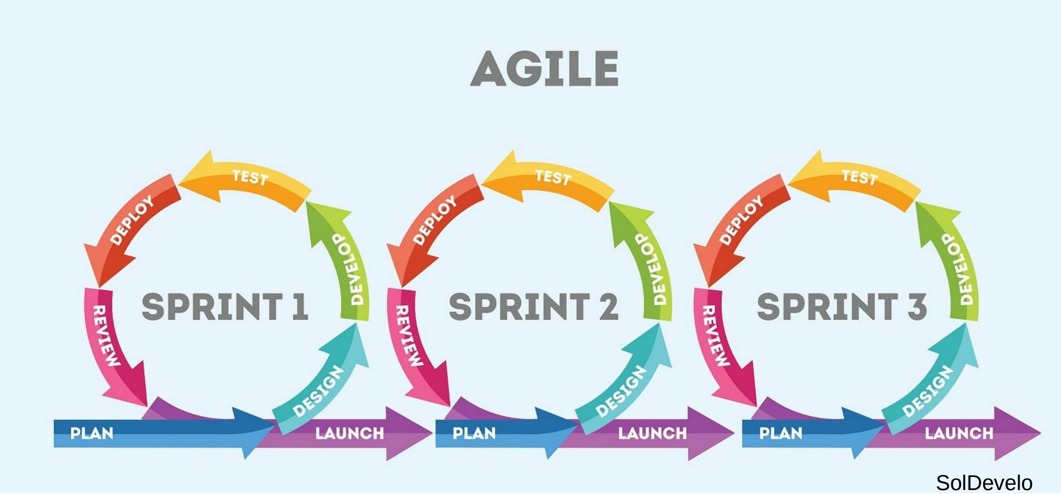
Need a way to iteratively test requirements!



bap-software.net

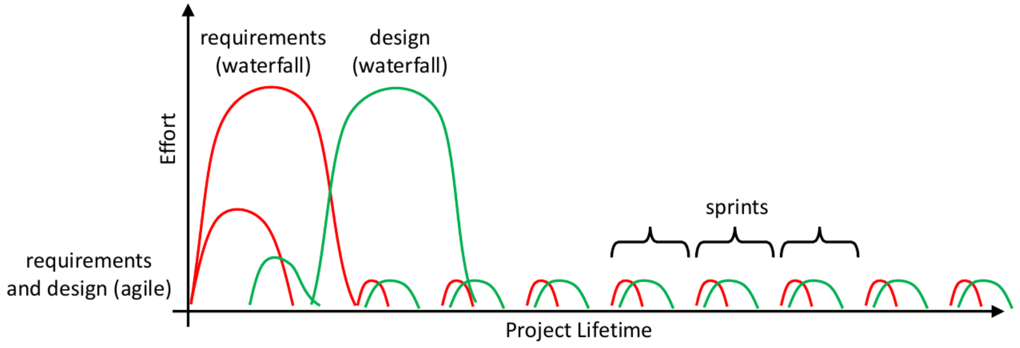
Agile

- *Iterative approach:* Each next step informs changes on previous step
- Value:
  - **Individuals and interactions** over processes and tools
  - **Working software** over comprehensive documentation
  - **Customer collaboration** over contract negotiation
  - **Responding to change** over following a plan



Waterfall vs. Agile

- **Waterfall** works well when:
  - Requirements don't change
  - Design is straight-forward/well understood
  - Developers are experienced in application area
- **Agile** works well when
  - Requirements are not well understood
  - Design is complex/challenging
  - Developers are new to the application



What is more common in your research?