Principles of Software Construction: Objects, Design, and Concurrency

Software Engineering for Teams

Charlie Garrod  Chris Timperley
Administrivia

• Homework 4c due today
  – Up to 75% of points lost on Homework 4a can be recovered by submitting revised design documents
• Homework 5 coming soon
  – Team signup sheet will be posted to Piazza today
  – Team signup deadline is next Tuesday
• Midterm in class next Thursday
  – Review session with Shruti and Alice
    Next Wednesday, 6-8pm, Hamerschlag B131* (provisional)
    There will be pizza! :-)

*provisional
<table>
<thead>
<tr>
<th>Intro to Java</th>
<th>UML</th>
<th>GUIs</th>
<th>More Git</th>
</tr>
</thead>
<tbody>
<tr>
<td>Git, CI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Part 1: Design at a Class Level</td>
<td>Part 2: Designing (Sub)systems</td>
<td>Part 3: Designing Concurrent Systems</td>
<td></td>
</tr>
<tr>
<td>Design for Change: Information Hiding, Contracts, Unit Testing, Design Patterns</td>
<td>Understanding the Problem</td>
<td>Concurrency Primitives, Synchronization</td>
<td></td>
</tr>
<tr>
<td>Design for Reuse: Inheritance, Delegation, Immutability, LSP, Design Patterns</td>
<td>Responsibility Assignment, Design Patterns, GUI vs Core, Design Case Studies</td>
<td>Designing Abstractions for Concurrency</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Testing Subsystems</td>
<td>Design for Reuse at Scale: Frameworks and APIs</td>
<td></td>
</tr>
</tbody>
</table>
Software engineering is inherently collaborative
“Understanding code is by far the activity at which professional developers spend most of their time.”

*Peter Hallam, C# language design team*

[https://blog.codinghorror.com/when-understanding-means-rewriting/](https://blog.codinghorror.com/when-understanding-means-rewriting/)
Design as communication

APIs, code, and documentation are the language that we use to **communicate** the intent of our software to other developers.

```java
public interface Collection<E> {
    // Ensures that collection contains o
    boolean add(E o);

    // Removes an instance of o from collection, if present
    boolean remove(Object o);

    // Returns true iff collection contains o
    boolean contains(Object o);

    // Returns number of elements in collection
    int size();

    // Returns true if collection is empty
    boolean isEmpty();

    ...  // Remainder omitted
}
```

https://www.uml-diagrams.org/examples/java-7-concurrent-executors-uml-class-diagram-example.png
When we get it right, code can read like poetry or a novel you don’t want to put down. But when we get it wrong, code can be □□□□□.
The group project will give you valuable experience of tackling real-world design and engineering challenges

- You have to write code that is understandable
- You have to work with other people’s APIs and code
- You have to read and understand other people’s code
  - It’s not enough for your code to produce the right result
- You will have to collectively make design decisions
- You will learn much more about good design

You will **communicate within a team** to develop plugins. You will also **communicate with other teams** to use plugins.

Our goal is to transform you from a programmer to an engineer.
Outline

- Group dynamics
- Tools and processes for software teams
Challenges of working as a team: Aligning expectations

• How do we make decisions?
  – Are you a *laissez-faire* or *by-the-book* person?
• ...
Challenges of working as a team: Aligning expectations

- How do we make decisions?
  - Are you a *laissez-faire* or *by-the-book* person?
- How do we divide the work?
  - Who should do what task?
  - Should team members be responsible for certain components?
  - How do we ensure a fair allocation of work?
- Does the group share the same goals and incentives?
  - Bad things can happen when these are misaligned!
- What happens when work isn’t done?
  - How will you make sure that the group stays on track?
- When do team members like to get work done?
  - Are they night owls or early birds? (Be honest!)
- What other commitments do your team members have?
- Where will you get the work done?
  - Will you work in the same location? Remotely? Asynchronously?
- ...
Team roles: You will probably have more than one

- **Facilitator**: Moderates team discussion and keeps the group on task.
- **Recorder**: Takes notes summarizing team discussions and decisions, and keeps all necessary records.
- **Timekeeper**: Keeps the group aware of time constraints and deadlines and makes sure meetings start on time.
- **Devil’s Advocate**: Raises counter-arguments and (constructive) objections; introduces alternative explanations and solutions.
- **Innovator**: Encourages imagination and contributes new and alternative perspectives and ideas.
- **Harmonizer**: Strives to create a positive team atmosphere.
- **Prioritizer**: Makes sure group focuses on most important issues and does not get caught up in unimportant details.
Running an effective meeting: Before the meeting

• Prepare an agenda
  – Beware of meetings without an agenda!

• Figure out a regular meeting slot
  – If there is no agenda, cancel the meeting (longer notice is better)
  – Make sure that your meeting has a clear cut-off time

• Determine a convenient place to meet
  – Minimum distractions; laptop space; whiteboards

• Be organised and prepared
Running an effective meeting: During the meeting

• At the start of the meeting, follow up on items from the previous meeting and *briefly* review status.
• At the end of the meeting, summarise the decisions that were made, the issues to be resolved, and the work to be done.
  – Agree on what each team member needs to do by the next meeting.
  – Set an agenda for the next meeting.
• **Every meeting should produce an artifact.**
  – The recorder should maintain minutes of the meeting.
  – A copy of the minutes should be sent to each team member.
  – Consider using a shared Google Drive or similar.
Aside: The importance of flow

“Unfortunately, you can’t turn on flow like a switch. It takes a slow descent into the subject, requiring 15 minutes or more of concentration before the state is locked in. During this immersion period, you are particularly sensitive to noise and interruption. A disruptive environment can make it difficult or impossible to attain flow.”

— Peopleware, Third Edition, Chapter 10
Outline

• Group dynamics
• Tools and processes for software teams
  – Identifying and assigning tasks
  – Collaborative development via GitHub
  – Testing strategies
Design the API

Basic Process:
(1) Determine minimal feature set
(2) Draw UML on the whiteboard.
(3) Sketch out your API on paper
(4) Write example code
(5) Review
(6) Repeat

// A collection of elements (root of the collection hierarchy)
public interface Collection<E> {
    // Ensures that collection contains o
    boolean add(E o);
    // Removes an instance of o from collection, if present
    boolean remove(Object o);
    // Returns true iff collection contains o
    boolean contains(Object o);
    // Returns number of elements in collection
    int size();
    // Returns true if collection is empty
    boolean isEmpty();
    ... // Remainder omitted
}
Break up tasks into GitHub Issues

Issues can represent both tasks and bugs that need to be fixed.

Issues should be:
- a *reasonable* chunk of work
- focused and cohesive
Is an issue too big? Break it up!

Add proxy for interacting with file system #7

ChrisTimperley commented on 25 Jan • edited

- #53 read contents of file
- #55 write to file
- append to file
- #181 implement find-like functionality
- copy file (i.e. shutil.copy)
- copy tree recursively (i.e. shutil.copytree)
- #47 chmod (i.e. os.chmod)
- #40 remove file (i.e. os.remove, rm)
- #41 remove directory (i.e. os.rmdir, rmdir)
- #44 remove directory and contents (i.e. shutil.rmtree, rm -r)
- #35 list contents of directory (i.e. os.listdir, ls)
- #36 create directory (non-recursive) (i.e. os.mkdir)
- #39 create directory (recursive) (i.e. os.makedirs)
- #34 check if file/directory exists (i.e. os.exists, os.isfile, os.isdir, os.islink)
- #48 touch (i.e. os.touch)
Break up tasks into GitHub Issues
Use labels to indicate priority and differentiate bugs from features.
Consider using milestones (e.g., HW5a, HW5b)

<table>
<thead>
<tr>
<th>Project</th>
<th>Due Date</th>
<th>Status</th>
<th>Open</th>
<th>Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiny Ticket Sprint</td>
<td>By October 25, 2019</td>
<td>16% complete</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>E Turtle Release</td>
<td>By November 22, 2019</td>
<td>63% complete</td>
<td>29</td>
<td>50</td>
</tr>
<tr>
<td>September</td>
<td>Past due by 22 days</td>
<td>81% complete</td>
<td>5</td>
<td>22</td>
</tr>
<tr>
<td>F Turtle Release</td>
<td>No due date</td>
<td>0% complete</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Now what?

<table>
<thead>
<tr>
<th>#</th>
<th>Title</th>
<th>Assignee</th>
<th>Issue Type</th>
<th>Priority</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Add Autolocalization as an action server</td>
<td></td>
<td>enhancement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Improved Recovery Node Behavior Tree</td>
<td></td>
<td>2 - Medium</td>
<td>enhancement</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Should we provide a node to TF Listeners?</td>
<td></td>
<td>1 - Low</td>
<td>enhancement</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Update system test to use NavigateToPose Action</td>
<td></td>
<td>2 - Medium</td>
<td>enhancement</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Implement waypoint following</td>
<td></td>
<td>3 - Low</td>
<td>enhancement</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Provide behavior tree current node on a ROS topic</td>
<td></td>
<td>help wanted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Slow robot if DWB missing performance requirements</td>
<td></td>
<td>2 - Medium</td>
<td>enhancement</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Map server providing dynamically-generated maps</td>
<td></td>
<td>2 - Medium</td>
<td>enhancement</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Maps from sub-map sources</td>
<td></td>
<td>2 - Medium</td>
<td>enhancement</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Multi-layer maps: map server and data structures design</td>
<td></td>
<td>2 - Medium</td>
<td>enhancement</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Need more extensive system tests</td>
<td></td>
<td>2 - Medium</td>
<td>enhancement</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Add Slow / safety zones to map</td>
<td></td>
<td>2 - Medium</td>
<td>enhancement</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Add support for Keep-out zones on map</td>
<td></td>
<td>2 - Medium</td>
<td>enhancement</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Map server design for multi-layer maps, multiple map sources, etc.</td>
<td></td>
<td>2 - Medium</td>
<td>enhancement</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Map server service to change map</td>
<td></td>
<td>2 - Medium</td>
<td>enhancement</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Time monitoring of time-sensitive tasks</td>
<td></td>
<td>2 - Medium</td>
<td>enhancement</td>
<td></td>
</tr>
</tbody>
</table>

---

17-214
Assigning issues to team members

• How do we assign issues to team members?
  – Split the issues equally?
  – Are all issues equally important and time consuming?
  – Should one person deal with all X-related matters?

• Unfair assignment can create resentment and bad dynamics.
  – Most of the time this happens, it’s unintentional!
  – Software engineers are really lousy at effort estimation.

https://jignashadesai.files.wordpress.com/2017/01/uneven.jpg
Beware: Humans aren’t good at estimating effort!

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.
— Douglas Hofstadter, Gödel, Escher, Bach: An Eternal Golden Braid

The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.
— Tom Cargill, Bell Labs
Use planning poker to estimate effort

https://www.mountaingoatsoftware.com/agile/planning-poker
When should we assign issues?

- No hard-and-fast rule.
- Self-assignment can be okay, and may be necessary.
- Need to maintain checks and balances! Review issue assignment during meetings.
How does a large software project get to be one year late?
One day at a time.

— Fred Brooks, *The Mythical Man-Month*

https://en.wikipedia.org/wiki/The_Mythical_Man-Month
Use a simple Kanban board to measure progress
Outline

• Group dynamics
• Tools and processes for software teams
  – Identifying and assigning tasks
  – Collaborative development via GitHub
  – Testing strategies
# Tackling the issues

<table>
<thead>
<tr>
<th>#</th>
<th>Summary</th>
<th>Status</th>
<th>Milestone</th>
</tr>
</thead>
<tbody>
<tr>
<td>151</td>
<td>Add Autolocalization as an action server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>152</td>
<td>Improved Recovery Node Behavior Tree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>153</td>
<td>Should we provide a node to TF Listeners?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>154</td>
<td>Update system test to use NavigateToPose Action</td>
<td></td>
<td></td>
</tr>
<tr>
<td>155</td>
<td>Implement waypoint following</td>
<td></td>
<td></td>
</tr>
<tr>
<td>156</td>
<td>Provide behavior tree current node on a ROS topic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>157</td>
<td>Slow robot if DBS missing performance requirements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>158</td>
<td>Map server providing dynamically-generated maps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>159</td>
<td>Maps from sub-map sources</td>
<td></td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>Multi-layer maps: map server and data structures design</td>
<td></td>
<td></td>
</tr>
<tr>
<td>161</td>
<td>Need more extensive system tests</td>
<td></td>
<td></td>
</tr>
<tr>
<td>162</td>
<td>Add Slow / safety zones to map</td>
<td></td>
<td></td>
</tr>
<tr>
<td>163</td>
<td>Add support for Keep-out zones on map</td>
<td></td>
<td></td>
</tr>
<tr>
<td>164</td>
<td>Map server design for multi-layer maps, multiple map sources, etc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>165</td>
<td>Map server service to change map</td>
<td></td>
<td></td>
</tr>
<tr>
<td>166</td>
<td>Time monitoring of time-sensitive tasks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Single-branch development doesn’t scale to teams
Use simple branch-based development

Create a new branch for each feature.

- allows parallel development
- no dealing with half-finished code
- no merge conflicts!

Every commit to “master” should pass your CI checks.
Use GitHub **pull requests** to review and merge changes

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.

Aside: You can also create **draft pull requests**

[Image of a computer screen showing the process of creating a draft pull request on GitHub]

Makes it easier for others to see progress without interrupting your workflow!

https://github.blog/2019-02-14-introducing-draft-pull-requests/
You’ve created your Pull Request. Now what?
When you merge your own pull request

Doing PRs
Better: Ask your teammates to review your pull request!

Tips for Code Review: Preparing your code for review

1. **Keep your changes small and cohesive**
   - Large pull requests (PRs) are tedious and difficult to review.
   - Try to avoid fixing multiple issues in one PR.

2. **Check your code before you submit your pull request**
   - Is the code readable? Do I need to add any comments?
   - Is the Travis build passing? If not, fix it.
   - Make life easier for the reviewer — things will move faster! :-)

3. **Don’t get too attached to your code**
   - You may have to change things. Don’t take it personally.
   - Be open to feedback — you’ll become a better engineer.
Tips for Code Review: Reviewing code

1. **Agree the purpose of review**
   - At a minimum, to see that the code is correct and tested.
   - Check for style, consistency, naming, etc.

2. **Be considerate**
   - Don’t review the author; review the code

3. **Provide constructive criticism**
   - Identify the *potential* issues and provide suggestions
   - If there’s any doubt, ask for clarification -- you may be wrong.
   - Don’t pass off your own opinions as fact

4. **Don’t change the pull request yourself**
   - At that point, you’re no longer a reviewer.

5. **Establish timeliness expectations**
   - How long should the PR author wait to receive a review?

https://alterconf.com/talks/unlearning-toxic-behaviors-code-review-culture
https://css-tricks.com/code-review-etiquette/
Bonus tip: Automatically close issues in commits/PRs

Use any of the following words:

- close #N, closes #N, closed #N
- fix #N, fixes #N, fixed #N
- resolve #N, resolves #N, resolved #N
Outline

• Group dynamics
• Tools and processes for software teams
  – Identifying and assigning tasks
  – Collaborative development
  – Testing strategies
Devise a testing strategy for your team

• Who should write tests?
  – The person responsible for the implementation?
  – Someone else?

• When should tests be written? When will tests be written?
  – Smaller issues = fewer tests
  – Write blackbox tests before implementation
  – Write additional whitebox tests before submitting a pull request

• Consider using mocks and (basic) dependency injection
  – Allow coupled components to be unit tested
  – Allows developers to work in parallel

https://site.mockito.org/
Summary

• Identify and discuss risks within your team
  – Get to know your teammates, and agree on your process
• Tools and services like Git, GitHub, and Trello can improve your engineering process and help your team work more effectively!
  – Some process is better than no process.
  – Use this advice as a set of guidelines; discover what works well for your team -- everyone is different.
• Check out the books below for much more on this topic!