CPSC 329
Software Development

Course Website

(not live yet – by the end of the day!)

- direct URL: http://math.hws.edu/bridgeman/courses/329/f26
- via my homepage: http://math.hws.edu/bridgeman
- via Canvas

Course Outline

Four key points are at the core of good software:

- fully understanding what the software is to do,
- planning and design before coding,
- proper use of modularity and abstraction in the design,
- and building in reliability through good coding practices and the integration of testing in the implementation process.

This course will address how to achieve those points. In addition, students will gain:

- an organized approach to software development;
- an understanding of the core principles of object-oriented design, and the ability to apply those principles to create flexible and maintainable software;
- a habit of good software, which includes coding standards, documentation, appreciation of the value of good design (and the willpower to refactor as needed to improve the design), and testing;
- experience with a substantial project, working in teams, and working with someone else's code;
- skills for life-long learning, specifically learning from examples, tutorials, and APIs;
- technical skills relevant to software development, including the use of integrated development environments, version control, and unit testing; and
- a basic technical understanding of some aspects of modern software such as GUIs, event-driven programming, user interface design, client-server networking, and threads.

Topics I

- OOP foundations
  - refresher on classes, objects, encapsulation
  - inheritance and polymorphism, abstract classes vs interfaces
- OO design
  - basic OO design principles
  - SOLID
  - core design patterns
  - UML class diagrams
- software construction practices
  - standards and defensive coding
  - code smells
  - refactoring

Topics II

- testing and verification
  - testing foundations and planning
  - test-driven development
  - JUnit
  - debugging techniques and tools
  - logging
- requirements
- professional practices and tools
  - teamwork
  - version control
  - working with other people's code and large codebases
  - professional ethics and intellectual property
  - reading documentation/APIs
  - career habits

Topics III

- GUI programming
  - event-driven programming
  - JavaFX: basics, controls, event handling, layout
  - UI design essentials
  - MVC
- concurrency and threads
  - threads in Java, thread life cycle
  - challenges: race conditions, synchronization, thread safety, deadlock
  - threads GUI programs
- networking and client-server
  - networking basics
  - Java sockets
  - client-server architecture

Course Logistics

- weekly tutorial-style labs introducing particular skills
- project arc – most phases are ~2 weeks
  - P0 (individual) – basic game with console (text-based) UI
  - P1a (pairs) – work in teams, add new rule variant(s) / additional features, testing
  - P1b (same pairs) – refactor according to design patterns, add additional rule variant(s) / additional features
  - P2a (new pairs) – add a GUI
  - P2b (same new pairs) – add threads to manage long-running background tasks
  - P3 (groups of three) – network play (client-server), identify new requirements
- 20-minute project handin meetings (individual) for most project phases
  - to talk through your submission and confirm that it genuinely reflects your own understanding and effort

Assessment and Grading

- engagement and professional presence – 15%
  - labs – feedback based on correctness, graded based on completeness
  - attendance – in class, lab, project handin meetings
    - for full credit, a maximum of three unexcused absences from class and one from lab are permitted
  - teamwork conduct – attending team meetings, being responsive, following through on commitments, communicating constructively
- midterm exams – 35% (approx. 11-12% each)
  - written, short answer-based focusing on foundational knowledge and underlying concepts
- projects – 50% (15% meeting specifications, 35% skills)
  - meeting specifications – deliverables satisfy the required criteria, project works correctly
  - skills – demonstrating that you can perform tasks and apply the foundational knowledge

Assessment and Grading

- reassessment
  - there are limited opportunities for reassessment – do the practice and respond to feedback *before* assessments (exams, project handin)
  - two reassessment opportunities
    - for the written exams, near the end of the course
    - for the projects, a take-home final exam (skills based) with an ~30-minute handin meeting during the scheduled final exam timeslot

AI Policy

- rationale
  - computer science is about the fundamentals, not only how to use the latest tools
  - academic environments are about learning for yourself – AI use shortcuts your thinking and inhibits your learning
  - many open source projects limit or prohibit AI use in contributions
  - AI is not responsible for the code – the developer is
  - ethical concerns about training data, bias, environmental concerns, etc

AI Policy

- AI use (generative, agentic, etc) is **not** permitted in work you turn in
  - no code generation, writing documents, generating ideas, automating tasks
  - no using AI to summarize or otherwise digest assignment handouts

- using AI as a peer tutor is **discouraged** unless specifically authorized
  - it can be valuable in helping with alternate explanations, generating practice problems, and providing feedback but it is also easy to slip into shortcutting think – and AI is not always correct in its answers

- search engine “AI overview” is acceptable in locating search results, but avoid using AI to shortcut the work of understanding sources and compiling information yourself
  - go to the links found and verify what it tells you!

Organized Approaches to Software Development

- small programs can survive an unplanned “just start coding” approach
- bigger systems can’t – a tangled codebase makes everything harder
  - requirements shift – change ripples unpredictably through the code
  - code gets reused by others – but isn’t cleanly separated from other parts of the codebase
  - mistakes get expensive to unwind – damage isn’t isolated
- an “organized approach” is set of habits that keep a project from collapsing under its own complexity

The Phases Every Approach Has to Cover

- **requirements** – what are we building, and for whom?
- **design** – how will it be structured?
- **construction** – actually building the software
- **testing/verification** – does it work, and did we build the right thing?
- **maintenance** – keeping it work as things change

- every approach to software development has to address these phases
  - the differences are about order and repetition

Waterfall

- do each phase once, in order
  - requirements → design → construction → testing → maintenance

- strengths
  - clear structure
  - easy to plan and staff
  - works well when requirements are stable and well-understood

- weaknesses
  - assumes you got the requirements and design right before writing any code – expensive to fix mistakes discovered later

Iterative and Incremental

- build in small, working slices
  - repeat the phases in small cycles, each producing a working (if partial) system
  - each cycle can incorporate what you learned from the last one

- strengths
  - catches bad assumptions early, while they are still cheap to fix

- weaknesses
  - requires discipline so that “iterative” doesn’t turn into “disorganized”

This Course

- we’ll be using an iterative and incremental approach
  - each phase of the project adds a new requirement or feature

- why?
  - mirrors the sequence of topics in the course content
  - creates checkpoints for feedback
  - test earlier design decisions against later changes
    - motivates techniques
  - real-world software development more commonly involves evolving a working system than building from scratch
  - modern industry practice is typically iterative/incremental