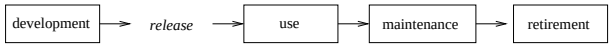
cis20.2
design and implementation of software applications 2
spring 2010
lecture # I.3

today's topics:

- software lifecycles
- software patterns
- agile processes
- extreme programming

software lifecycles

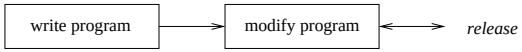
- software is not a build-one-and-throw-away process
- that's far too expensive
- we need to implement a *process* so that software is maintained correctly
- this is called the *software life cycle*:



- there are several *process models*:
  - *build-and-fix model*
  - *waterfall model*
  - *iterative model*
  - *evolutionary model*

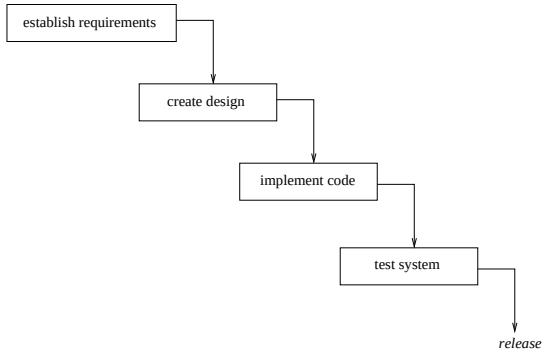
build-and-fix model

- the oldest model
- probably what you've been doing when you write your homework...



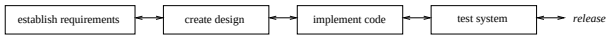
waterfall model

- developed as software evolved into large projects, involving many lines of code, many files and many programmers working together on the same large project...



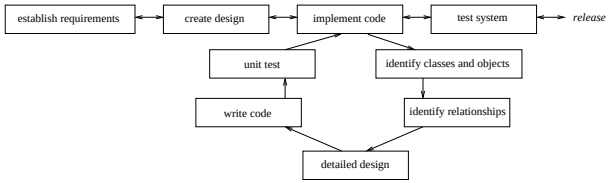
iterative model

- developed after it was recognized that the waterfall model was unrealistic
- each step can be (and usually is) revisited
- especially common in large companies, where multiple people are working on the same project and the people who, for example, "establish requirements" are not the same people who "create design" or "implement code" or "test system"



evolutionary model

- evolved, again from companies where large software projects are developed and maintained, particularly after the introduction of the "object-oriented" way of thinking
- emphasizes *modularity* and allows for software re-use as well as testing of individual modules to make sure that each piece is robust and correct before it is added to the whole



design patterns: overview

- one of the "hot topics" in the object-oriented software engineering community
- goal: to create a body of solutions to common problems in the area of software development
- this includes common vocabulary, strategies/algorithms and code for re-use
- origins: **Design Patterns: Elements of Reusable Object-Oriented Software**, by Gamma, Helm, Johnson and Vlissides
  also called the "Gang of Four" or **GoF**
- history:
  - initially used in Smalltalk to help novice programmers, as a "pattern language"
  - later used in C++ as an "idiom"
  - these ideas evolved into "design patterns"

design patterns: definition

- "A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts." [Riehle and Zullinghoven, 1996]
- in software terms:
  "A pattern is a named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces." [Appleton, 2000]
- usually involve a modular architecture which is comprised of parts which together make a whole; the patterns come in when constructing the modules
- a pattern is a three-part rule containing: *context*, *problem* and *solution*
  or a pattern is a "thing" that happens in the world, the rule which tells how to create the thing and when to create it [Gabriel]

types of patterns

- generative patterns are used to create something
- non-generative patterns are used to describe something that recurs, but don't tell how to create it
- generative patterns show how to create something and illustrate characteristics of good (best) practice
- everything isn't a pattern!
- a pattern must have the three parts (context, problem, solution) and *it must recur!*
- good patterns:
  - solve a problem
  - demonstrate a proven concept
  - provide a non-obvious solution
  - describe a relationship between modules and system structures
  - contain a significant human component

- a pattern is not a "lesson learned"
- a pattern is a "best practice"
- we focus here on *software design patterns*, though many other types of patterns exist (like organizational patterns, analysis patterns, etc)

elements of a pattern

- "Alexandrian form"

```
IF   you find yourself in CONTEXT
     for example EXAMPLES,
     with PROBLEM,
     entailing FORCES
THEN for some REASONS,
     apply DESIGN FORM AND/OR RULE
     to construct SOLUTION
     leading to NEW CONTEXT and OTHER PATTERNS
```

- contain the following essential elements:
  - **name** — meaningful name for the pattern; can include a classification
  - **problem** — statement of the problem and its intent (goals and objectives it wishes to obtain)
  - **context** — preconditions under which problem and solution recur; i.e., applicability of the pattern

  - **forces** — description of relevant forces and constraints
  - **solution** — static relationships and dynamic rules describing how to realize the desired outcome
  - **examples** — sample applications of the pattern
  - **resulting context** — state of system after pattern has been applied
  - **rationale** — justifying explanation of steps/rules in the pattern
  - **related patterns** — relationships between this pattern and others in the same pattern language/system
  - **known uses** — known occurrences of the pattern and its aplicaiton within existing systems; may overlap with examples, but may be more complex since "examples" should be simple

forces

- generalize the kinds of criteria that software engineers use to justify designs and implementations
- e.g., in algorithms, the main force to be resolved is efficiency (time complexity)
- but patterns deal with the larger, harder-to-measure, and conflicting sets of goals and constraints encountered in the development of every artifact created
- examples:
  - Correctness
    - Completeness and correctness of solution
    - Static type safety, dynamic type safety
    - Multithreaded safety, liveness
    - Fault tolerance, transactionality
    - Security, robustness
  - Resources
    - Efficiency: performance, time complexity, number of messages sent, bandwidth requirements

    - Space utilization: number of memory cells, objects, threads, processes, communication channels, processors, ...
    - Incrementalness (on-demand-ness)
    - Policy dynamics: Fairness, equilibrium, stability
  - Structure
    - Modularity, encapsulation, coupling, independence
    - Extensibility: subclassibility, tunability, evolvability, maintainability
    - Reusability, openness, composibility, portability, embeddability
    - Context dependence
    - Interoperability
    - ... other "ilities" and "quality factors"
  - Construction
    - Understandability, minimality, simplicity, elegance.
    - Error-proneness of implementation
    - Coexistence with other software
    - Maintainability
    - Impact on/of development process
    - Impact on/of development team structure and dynamics

    - Impact on/of user participation
    - Impact on/of productivity, scheduling, cost
  - Usage
    - Ethics of use
    - Human factors: learnability, undoability, ...
    - Adaptability to a changing world
    - Aesthetics
    - Medical and environmental impact
    - Social, economic and political impact
    - ... other impact on human existence"

qualities of patterns

- encapsulation and abstraction
  - should encapsulate a well-defined problem
  - should abstract domain knowledge and experience
- openness and variability
  - should be open for extensions, in a wide variety of applications
- generativity and composability
  - applying one pattern should generate the context for another...
- equilibrium
  - should achieve a balance between forces and constraints

frameworks

- closely related to patterns
- design patterns can be used by frameworks but are more abstract than frameworks
- design patterns are smaller architecture elements than frameworks
- design patterns are more general than frameworks
- frameworks use "inverted flow of control" between its clients and itself
  "don't call us, we'll call you"...
  "leave the driving to us"...

patterns: cataloging and writing

- pattern catalog: collection of related patterns
- pattern system: set of related patterns with an underlying structure connecting the patterns together
- a pattern system is more structured than a pattern catalog, which is more like a list
- "pattern mining": looking for patterns in an existing system
- writing patterns is HARD
- patterns are not a silver bullet!

Agile processes

- *the Agile Alliance* — group of industry folks who gathered in 2001
- agile values:
  - individuals and interactions over processes and tools
  - working software over comprehensive documentation
  - customer collaboration over contract negotiation
  - responding to change over following a plan
- principles
  - Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
  - Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
  - Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.
  - Business people and developers must work together daily throughout the project.

  - Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
  - The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
  - Working software is the primary measure of progress.
  - Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
  - Continuous attention to technical excellence and good design enhances agility.
  - Simplicity—the art of maximizing the amount of work done—is essential.
  - The best architectures, requirements, and designs emerge from self-organizing teams.
  - At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

extreme programming

- an agile methodology
- set of *core practices*
  - whole team
  - planning game, small releases, customer tests
  - simple design, pair programming, test-driven development, design improvement
  - continuous integration, collective code ownership, coding standard
  - metaphor, sustainable pace