Non-object-oriented design methods

Software Requirements and Design
CITS 4401
Lecture 15

(reminder) Software Design is

- a creative process
  - no cook book solutions
- goal driven
  - we create a design for solving some problem
- constraint driven
  - by the function to be served and the constructions which are possible
- good designs can be recognised
  - simple, coherent, adequately meets requirements, adaptable

Some Other Design Methods

- So far we have focussed on object-oriented design
- However OO design is NOT suitable for all types of systems
- All design methods involve a hierarchy of decompositions which partition the design into subsystems or components
- This lecture outlines some non-OO methods for doing this decomposition

Data-Oriented Design (DOD)

- A school of thought: “identification of the inherent data structure can be used to derive the structure of a program”
- Logical construction of programs (LCP)
  - Developed by Warnier (‘74)
  - Draw upon relationship between data structure and procedure structure
- DOD may be successfully applied in applications that have well-defined, hierarchical structure of information
**Event-Oriented Design**

- Each subsystem consists of components that handle similar type of events
- Examples: editors; rule-based systems in AI; most real-time systems are event driven
- Disadvantage:
  - Subsystems don’t know if or when events will be handled

**“Structured” design**

- Try to decompose each subsystem into modules
- Two main strategies:
  1. **Object-oriented decomposition** – subsystem decomposed into a set of communicating objects
  2. **Function-oriented pipelining** – subsystem decomposed into functional modules

**In summary…**

- Data-oriented design
  - start with external data structures and continue by adding more detailed data structures

- Event-oriented design
  - what events are possible for this system?
  - what response is required for each event?
  - how does each event change the system state?

**Structured design**

- modular / functional design
  - each subsystem captures one of the functions of the system

- outside-in / top-down design
  - start with black boxes and their inputs and outputs
  - then divide each box into internal input-output boxes

**In summary…**

- Structured design
  - modular / functional design
  - outside-in / top-down design
Outside-In Design Example

Brick Sorter Design

- Input via the light sensor, sends message to the controller
- Output via the kick arm, receives eject warning from the sensor process and then kicks when the controller is ready
- Controller manages the timely co-ordination of the inputs and outputs
- Example: presented using the Uppaal system
  - see www.uppaal.com for more information

Verification

- Eventually, the controller is ready to eject a red brick
  - E<> Controller.c3
- Eventually bricks reach the end of the conveyor
  - E<> Box.k5
- The observer never sees a red brick (because they have all been correctly removed) and so he is never sad
  - A[\]not Observer.sad

E <> S means “some state satisfying S should be reachable”
A[] S means “S should be true in all reachable states”
Formal Methods and Refinement

- A final (extreme) alternative to the design process is the use of formal methods.
- Software specifications are meticulously transformed into mathematical statements.
- Then a process of refinement is used to derive provably working code from the mathematical specification.

Formal Methods cont.

- Formal methods are used for safety critical applications.
- Zed is a specification language that can map specifications in first-order logic into executable pseudo-code.
- The requirements stage of development is very expensive, but the design, implementation and testing can be a lot cheaper than other methodologies.

Agile Methods

- In the 80s and early 90s, a widespread view that the best way to achieve better software was through careful project planning, formalized quality assurance...
  - Suitable for large critical projects shared by large teams (mostly located at different locations)
  - For medium to small sized projects, the overhead is too large
- Agile methods rely on an iterative approach to software specifications, development and delivery. They are designed for business applications where the system requirements usually change rapidly.

Principles of Agile Methods

All agile methods share the following set of principles:

1. **Customer involvement**
   - Customers should be closely involved throughout the development process. Their role is to provide and prioritise new system requirements and evaluate the iterations of the system.

2. **Incremental delivery**
   - The software is developed in increments with the customer specifying the requirements to be included in each increment.

3. **People, not process**
   - The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Principles of Agile Methods (cont.)

4. Embrace change
   - Expect the system requirements to change, so design the system to accommodate these changes

5. Maintain simplicity
   - Focus on simplicity in both the software and the process. Wherever possible, actively work to eliminate complexity from the system.

Advantages of Agile Methods

- Better customer satisfaction by the rapid and continuous delivery of software.
- People and interactions are emphasized rather than process and tools. Customers, developers and testers constantly interact with each other.
- The close interaction between customer and software developer allows continuous attention to technical excellence and good design.
- Software development is able to adapt to changing circumstances. Even late changes in requirements are welcomed.

Disadvantages of Agile Methods

- In case of some software deliverables, especially the large ones, it is difficult to assess the effort required at the beginning of the software development life cycle.
- There is lack of emphasis on necessary designing and documentation.
- The project can easily get taken off track if the customer representative is not clear what final outcome that they want.

Best-known Agile Methods

- **Extreme Programming**
  - Commonly abbreviated as XP
- **Scrum**
  - Proposed by Schwaber and Beedle (2001)
- **Crystal Clear**
  - Proposed by Cockburn (2001)
- **Adaptive Software Development**
  - Proposed by Highsmith (2000)
- **Feature Driven Development**
  - Proposed by Palmer and Felsing (2002)
- **Test Driven Development (TDD)**
  - Sometimes referred to as test-driven design
Extreme Programming (XP)

**Extreme programming**
- emphasizes on customer satisfaction
- improves software project on 5 essential ways: **communication, simplicity, feedback, respect,** and **courage**
- advocates frequent "releases" in short development cycles

Other elements of XP include:
- Pair programming
- Extensive code review
- Unit testing
- Avoid programming of features until they are actually needed

Scrum

- **Scrum** defines a flexible, holistic product development strategy where a team works as a unit to reach a common goal.
- It encourages team members to self-organize, to be physically co-located, and close online collaboration.
- It adopts an **empirical** approach to requirements changes by maximizing the software team’s ability to deliver the product quickly.
- There are 3 core roles in the Scrum framework:
  1. Product owner (representing the stakeholders)
  2. Project team
  3. Scrum master
- Other elements in the framework: **sprint** (a basic unit of development in Scrum), **daily scrum**, **sprint review** and **sprint retrospective**

Crystal

- **Crystal** methodology focuses on **people, interaction, community, skills, talents,** and **communications.** The software process is consider important, but secondary.
- **Crystal Clear**
  - is a member of the Crystal family of methodology
  - is considered an example of an agile
  - can be applied to small team (up to 8 developers) co-located working on systems that are not life critical.
  - has the following properties:
    - Frequent delivery of usable code to client/users
    - Reflective improvement
    - Face-to-face close communication preferably by being co-located

Adaptive Software Development

- Adaptive Software Development (ASD) embodies the principle that developers should continuously adapt the software process to the work at hand.
- ASD replaces the traditional waterfall cycle with a repeating series of **speculate, collaborate,** and **learn** cycles.
  - **Speculation** – during speculation, the project is initiated and adaptive cycle planning is conducted, the set of release cycles is defined.
  - **Collaboration** – refers to the efforts for balancing the work based on the predictive parts of the environment and the uncertain surrounding mix of changes.
  - **Learning** – during the learning cycle, knowledge is gathered by making small mistakes based on false assumptions and correcting those mistakes. This leads to greater experience and eventually mastery in the problem domain.
Feature Driven Development

- FDD blends a number of industry-recognized best practices together.
- Its main purpose is to deliver tangible, working software repeatedly in a timely manner.
- FDD is a model-driven short-iteration process that consists of five basic activities:
  1. **Develop overall model** – high-level walkthrough of the scope of the system; creation of detailed domain model.
  2. **Build feature list** – The knowledge gathered during the initial modelling above is used to identify a list of features (a feature is a small client-valued function for the system).
  3. **Plan by the feature** – produce the development plan from the feature list; assign ownership of feature sets to programmers
  4. **Design by feature** – produce a design package for each feature; a chief programmer develops a selected set of features within 2 weeks.
  5. **Build by feature** – After a unit test and a successful code inspection, the completed feature is promoted to the main build.

Test Driven Development

- Test driven development is an **Agile**
  - Individuals and interactions over processes and tools
  - Working software over comprehensive documentation
  - Customer collaboration over contract negotiation
  - Responding to change over following a plan
  - TDD maps the requirements directly into testing code. The source code is then written specifically to pass these tests (and only to pass these tests)
  - This process is repeated incrementally until the product passes all the tests, and thus meets the requirements.

SRD Road Map

- This is the final lecture in the series on software design
- Next lecture onward we will study software requirements engineering

---

Scott Ambler
References

  - Section 17.1 “Agile Methods”
  - Section 17.2 “Extreme Programming”