Principles of Software Construction: Objects, Design and Concurrency

Introduction to Design

Christian Kästner

Charlie Garrod
The four course themes

- **Threads and Concurrency**
  - Concurrency is a crucial system abstraction
  - E.g., background computing while responding to users
  - Concurrency is necessary for performance
  - Multicore processors and distributed computing
  - Our focus: application-level concurrency
  - Cf. functional parallelism (150, 210) and systems concurrency (213)

- **Object-oriented programming**
  - For flexible designs and reusable code
  - A primary paradigm in industry – basis for modern frameworks
  - Focus on Java – used in industry, some upper-division courses

- **Analysis and Modeling**
  - Practical specification techniques and verification tools
  - Address challenges of threading, correct library usage, etc.

- **Design**
  - Proposing and evaluating alternatives
  - Modularity, information hiding, and planning for change
  - Patterns: well-known solutions to design problems
Learning Goals

• What is software design?
• Making trade offs
• Applying the modeling process from domain model to object model to implementation
• Basic modeling with UML (static and dynamic)
• Separating the different levels of UML use
Goal of Software Design

- For each desired program behavior there are infinitely many programs that have this behavior
  - What are the differences between the variants?
  - Which variant should we choose?

- Since we usually have to synthesize rather than choose the solution...
  - How can we design a variant that has the desired properties?
void sort(int[] list, String order) {
    ...
    boolean mustswap;
    if (order.equals("up")) {
        mustswap = list[i] < list[j];
    } else if (order.equals("down")) {
        mustswap = list[i] > list[j];
    }
    ...
}
void sort(int[] list, Comparator cmp) {
    ... 
    boolean mustswap;
    mustswap = cmp.compare(list[i], list[j]);
    ... 
}

interface Comparator {
    boolean compare(int i, int j);
}

class UpComparator implements Comparator {
    boolean compare(int I, int j) { return I < j; }
}

class DownComparator implements Comparator {
    boolean compare(int I, int j) { return I > j; }
}

(by the way, this design is called “strategy pattern”)

void sort(int[] list, String order) {
    boolean mustswap;
    if (order.equals("up")) {
        mustswap = list[i] < list[j];
    } else if (order.equals("down")) {
        mustswap = list[i] > list[j];
    }
    ...
}

void sort(int[] list, Comparator cmp) {
    boolean mustswap;
    mustswap = cmp.compare(list[i], list[j]);
    ...
}

interface Comparator {
    boolean compare(int i, int j);
}

class UpComparator implements Comparator {
    boolean compare(int I, int j) { return i<j; }
}

class DownComparator implements Comparator {
    boolean compare(int I, int j) { return i>j; }
}
Quality of a Software Design

• How can we measure the internal quality of a software design?
  ▪ Extensibility, Maintainability, Understandability, Readability, ...
  ▪ Robustness to change
  ▪ Low Coupling & High Cohesion
  ▪ Reusability
  ▪ Testability
  ▪ => modularity

• ...as opposed to external quality
  ▪ Correctness: Valid implementation of requirements
  ▪ Ease of Use
  ▪ Resource consumption
  ▪ Legal issues, political issues, ...
The bad news
it depends
(see context)

depends on what?
what are scenarios?
what are tradeoffs?
"Software engineering is the branch of computer science that creates practical, cost-effective solutions to computing and information processing problems, preferentially by applying scientific knowledge, developing software systems in the service of mankind. Software engineering entails making **decisions** under constraints of limited time, knowledge, and resources. [...] 

Engineering quality resides in engineering judgment. [...] Quality of the software product depends on the engineer's faithfulness to the engineered artifact. [...] Engineering requires reconciling conflicting constraints. [...] Engineering skills improve as a result of careful systematic reflection on experience. [...] Costs and time constraints matter, not just capability. [...] 

Software Engineering for the 21st Century: A basis for rethinking the curriculum Manifesto, CMU-ISRI-05-108
Design Strategies

• Design while coding
  ▪ Implement a solution
  ▪ Try-revise, ideally supported by refactorings

• Draw, then code
  ▪ Draw some diagrams
  ▪ Explore solution at a higher level
  ▪ Explore alternatives
  ▪ Then switch to coding
  ▪ Iterate if necessary
  ▪ Discuss: Worth the overhead? How much drawing?

• Draw only
  ▪ Generate code from diagrams
  ▪ Diagrams turn into programming language
  ▪ Requires very formal process of using diagramming languages
The design process

1. **Object-Oriented Analysis**
   - Understand the problem
   - Identify the key **concepts** and their relationships
   - Build a (visual) vocabulary
   - Create a **domain model** (aka conceptual model)

2. **Object-Oriented Design**
   - Identify **software classes** and their relationships with **class diagrams**
   - Assign responsibilities (attributes, methods)
   - Explore **behavior** with **interaction diagrams**
   - Explore design alternatives
   - Create an **object model** (aka design model and design class diagram) and **interaction models**

3. **Implementation**
   - Map designs to code, implementing classes and methods
UML: A visual modeling language

• Unified Modeling Language

• Graphical Notation to describe classes, objects, behavior, and more

• You will need:
  ▪ Class Diagrams
  ▪ Interaction Diagrams (Sequence or Communication Diagrams)

• Read chapter 2 of the textbook "Design Patterns Explained"

• Additional readings:
  ▪ Extra slides on 214 website
  ▪ Craig Larman, Applying UML and Patterns, Prentice Hall, 2004
  ▪ Martin Fowler. UML Distilled. Addison-Wesley, 2003
Code vs Graphical Models

- Graphical models abstract from implementations
- Focus on interfaces and relationships, or on interactions, ... while hiding details
  - Easier to communicate
  - Allows focus on higher-level design issues
- Forces to make relationships explicit
- Useful for sketching, trying alternatives, and documentation
Demo / Discussion

Virtual World
A word on notation

• UML notation is broadly known, well documented

• Informal notations/sketching often sufficient, but potentially ambiguous for communication and documentation

• In practice:
  ▪ Graphical modeling very common in general
  ▪ Agree on some notation
  ▪ Adapt/extend as needed
  ▪ UML rarely full heartedly adopted

• In this course
  ▪ Use UML and conventions for communication
  ▪ Keep it simple
  ▪ Clarity is imperative, document your extensions/shortcuts
  ▪ We don't require or recommend a drawing tool
Aside: UML in Practice

- No UML (35/50)
- Retrofit (1/50)
  - don’t really use UML, but retrofit UML in order to satisfy management or comply with customer requirements;
- Automated code generation (3/50)
  - UML is not used in design, but is used to capture the design when it stabilizes, in order to generate code automatically (typically in the context of product lines);
- Selective (11/50)
  - UML is used in design in a personal, selective, and informal way, for as long as it is considered useful, after which it is discarded;
- Wholehearted (0/50 – but described in secondary reports)
  - organizational, top-down introduction of UML, with investment in champions, tools and culture change, so that UML use is deeply embedded.

Further Reading: Petre. UML in Practice. ICSE 2013
Aside: UML in Practice

• Reasons for No Use
  - Lack of Context
  - Overheads of Understanding the Notation
  - Issues of Synchronization/Consistency

• Selective Use
  - UML as "Thought Tool"
  - Communication with Stakeholders
  - Collaborative Dialogues (eg. with architects)
  - Significant Adaptation
  - "Keeping it small"

Further Reading: Petre. UML in Practice.

<table>
<thead>
<tr>
<th>UML diagrams</th>
<th>Number of users</th>
<th>Reported to be used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class diagrams</td>
<td>7</td>
<td>structure, conceptual models, concept analysis of domain, architecture, interfaces</td>
</tr>
<tr>
<td>Sequence diagrams</td>
<td>6</td>
<td>requirements elicitation, eliciting behaviors, instantiation history</td>
</tr>
<tr>
<td>Activity diagrams</td>
<td>6</td>
<td>modeling concurrency, eliciting useful behaviors, ordering processes</td>
</tr>
<tr>
<td>State machine diagrams</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Use case diagrams</td>
<td>1</td>
<td>represent requirements</td>
</tr>
</tbody>
</table>
Aside: Diagramming in Practice


<table>
<thead>
<tr>
<th>Transient</th>
<th>Understand</th>
<th>Design</th>
<th>Communicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reiterated</td>
<td>1) Understand</td>
<td>3) Refactor</td>
<td>5) Onboarding</td>
</tr>
<tr>
<td></td>
<td>2) Ad-hoc</td>
<td></td>
<td>6) Secondary</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>stakeholders</td>
</tr>
<tr>
<td>Rendered</td>
<td></td>
<td>4) Design review</td>
<td>7) Customer</td>
</tr>
<tr>
<td>Archival</td>
<td></td>
<td></td>
<td>8) Hallway art</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9) Documentation</td>
</tr>
</tbody>
</table>
Aside: Diagramming in Practice

Aside: Key Observations

• How used?
  ▪ transient forms for exploration, permanent solutions for communication with larger groups
  ▪ mostly ad-hoc white-board diagrams during meetings

• Why used?
  ▪ to understand, to design, to communicate
  ▪ "code is king"

• Graphical conventions?
  ▪ Use of formal diagramming language is low
  ▪ too formal for mostly informal visualizations; cost benefit ratio

• Culture?
  ▪ limited adoption of drawing tools;
  ▪ high value diagrams recreated more formally
Literature on OO Design

• Alan Shalloway and James Trott. Design Patterns Explained, Addison Wesley, 2004
  ▪ Brief introduction to UML
  ▪ Introduction to design with design patterns
  ▪ Mandatory reading

• Craig Larman, Applying UML and Patterns, Prentice Hall, 2004
  ▪ Introduction to UML
  ▪ Excellent discussion of object-oriented analysis and object-oriented design with and without patterns
  ▪ Detailed additional material, many guidelines

• Bertrand Meyer, Object-Oriented Software Construction, Prentice Hall, 1997
  ▪ Detailed discussion of design goals and modularity