Software Engineering Lecture Notes

September 4, 2019

Contents

Preface

These notes reflect the content of an ongoing Software Engineering course delivered at the Computer Science Department of the School of Science and Technology of Nazarbayev University. The course is required for Computer Science majors and has evolved with each iteration.

Software engineering is a critical part of the Computer Science curriculum. Our most fundamental aim is to transmit an appreciation for a skillful approach to software development. Moreover, we strive to emphasize both the technical and human aspects of software development.

How to read these notes

These notes are intended as a *companion* to the textbook [1], the lectures, and other assigned readings. You should think of these notes as guide. It is incumbent upon you to follow up with required readings and activities. There are required readings associated with each chapter. These readings are either listed in the body of the document or in margin notes. Be mindful of this because, in addition to the contents of these notes, you are responsible for knowing the lectures, exercises, and assigned readings.

Also, in these notes, *nonterminal* symbols shall be written, typically, in angle brackets `<nonterminal>`.

Chapter 1

Introduction

The goal of this section is to define the basic concepts of software engineering and outline the topics that will be covered. The course is Java-based and assumes proficiency with the language. Where appropriate, coding topics will be reviewed. As a refresher in Java (or to fill any knowledge gaps) the student is encouraged to work through Oracle's preparatory materials for the Java SE I and II programmer examinations [2]. Working knowledge of that content is assumed. The primary textbook for the course is "Applying UML and Patterns" by Craig Larman [1].

1.1 The Software Crisis

Required Readings — El Emam and Koru [3], Laitinen *et al* [4]

The "Software Crisis" or the "Crisis of Software" is a common name for the general difficulty of developing complex software systems. It can also refer to an observation that the capabilities of software often lag behind those of the hardware [5]. These problems were recognized in the late 1960s when significantly more powerful computing hardware started to become available. In fact, one of the earliest uses of the term "Software Engineering" comes from a 1968 conference sponsored by NATO [6]. Margaret Hamilton, a scientist who worked on early NASA missions, is also credited with coining the term. In the 1960s and 1970s, clients for software were primarily large-scale defense projects and it is these types of projects that shaped our initial understanding of software engineering (this is discussed in the first chapter of [7]). Advances made during this time resulted in many of the conveniences that are, today, easily taken for granted such as high-level languages and structured programming. However, it is also true that the industry has been evolving. Consider the following from Laitinen *et al* [4].

> Software engineering originally targeted large, contract-based development for defense and industry, and the discipline has achieved significant improvement in that area. Although contract software is no less important than it was, there are so many more types

> and modes of development that deserve serious investigation from a software engineering standpoint. (Laitinen *et al*, 2000 [4])

Although much progress has been made in the decades since the first electronic computers were invented, the central problems of software engineering remain unsolved. We continue to witness the failure of high-profile software projects. In this context, failure can mean a variety of things including project cancellation, lost investment, missed deadlines, cost overruns, or lack of acceptance by the users. In "mission critical" systems, software failure can even mean the loss of life.

We can also observe that the community is continually producing new solutions. For example, consider Deloitte's dizzying map of the "Agile Landscape"[1].There is always something out there that is being promised as the cure to the Software Crisis. One of the biggest pitfalls in software engineering is to chase blindly after all the latest fads.

The practices, principles, and values of software engineering are meant to address the difficulties of developing complex software systems. But what is software engineering exactly? The IEEE, conveniently, provides a definition in one of its standards.

> The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.
> (IEEE Std 610.12-1990 Definition of Software Engineering)

To be fair, the plain text of this definition comes across as circular and not particularly informative. To get the entire effect, the reader should make some effort to read between the lines. Construction can be a helpful metaphor for software engineering (for example, building a bridge, a dam, or a skyscraper). Consider the planning involved in undertaking a large construction project. If you were faced with the management of such a task, what questions would you ask? How, you may ask, will the project manage the coordination of people and materials? Also, consider the difference between the individual and a large team. Working alone, a skilled individual might alter an existing building or make a modest structure. On the other hand, a team working together can remake the entire landscape. The situation for software is analogous. Software engineering is about honing your skills, gaining confidence as a developer, and learning to integrate into a larger team for the purpose of creating something that is beyond what you can accomplish alone.

In the influential book *Extreme Programming Explained* the authors describe their philosophy of software engineering as follows,

> Good relationships lead to good business. Productivity and confidence are related to our human relationships in the workplace as

[1] Chris Webb, `http://blog.deloitte.com.au/navigating-the-agile-landscape/`

> well as to our coding or other work activities. You need both technique and good relationships to be successful.
>
> (Andres and Beck, 2004 [8])

We also find this sentiment articulated expertly in the Management literature. Consider the following quote from Peter Drucker.

> Management is about human beings. Its task is to make people capable of joint performance, to make their strengths effective and their weaknesses irrelevant. This is what organization is all about, and it is the reason that management is the critical determining factor. (Drucker [9], quoted by Lanham [10])

There is a fundamental *human* element to success in software engineering.

Undoubtedly, there is considerable need today for capable software engineers. Human life is dominated by technology and, with the rise in social media and the nascent era of "Big Data" [11, 12], it is hard to imagine a reversal in that trend. Without good software engineering, the profession will turn on both its practitioners and clients and will increasingly hold society hostage to its potential for failure.

In the following section, we compile a list of notable software failures (from a variety of spheres) to further motivate software engineering.

1.1.1 Software Failures

In this section, we recount examples of notable failures in software engineering. This list is not meant to be exhaustive.

healthcare.gov

Required Readings — Cleland-Huang [13]

In 2010, the United States Congress passed the Patient Protection and Affordable Care Act (often called the ACA or "Obamacare"). This was the signature legislative achievement of the Obama administration. However, despite the assurances of the administration that the online component of the ACA would make obtaining health insurance "as easy as ordering a pizza", the *healthcare.gov* website experienced a disastrous rollout [13]. This failure nearly derailed the legislation itself and left congresspeople wondering if the government was simply incapable of creating a website. During congressional hearings, the software contractors, in particular, came in for heavy criticism.

The launch of the site was widely seen as a disaster. Those responsible did not correctly anticipate the loads that would incur at launch time. And even users who were able to access the site found it to be non-functional with serious design flaws.

Not surprisingly, because the law itself was already controversial, there was great scrutiny on the *healthcare.gov* rollout. Ultimately, this event cast the profession in an unflattering light.

Required Readings — Oberg [14]

Mars Climate Orbiter

The Mars Climate Orbiter is an example of a software failure impacting a scientific endeavor. In 1998, NASA launched the Mars Climate Orbiter. This vehicle was supposed to orbit the planet Mars and collect data on the climate and surface of the planet [14].

However, a ground computer system designed by an outside contractor (Lockheed Martin) produced results in American units while the spacecraft expected quantities in metric units. As a result of this error, the entry into Martian orbit was miscalculated and the orbiter burned up in the atmosphere. In total, the project had cost approximately 330 million dollars.

An event like this is important because it can affect public attitudes towards taxpayer funded science.

Numerous Video Games

Consider the following quote from McBreen.

> For some software, rapid development of feature-rich applications is what matters. The idea is that users will put up with errors in programs because they have so many useful features that are unobtainable elsewhere. (McBreen, 2001 [7])

For McBreen, this trade-off is a logical consequence of following an "engineering" mentality with software development (he, instead, is a proponent of the *craftsmanship* model). In any event, the quote could easily describe today's video game industry. In the past decade, entertainment software has moved from a *retail* model to a *digital download* model [15]. This is witnessed by the success of online marketplaces such as Valve Corporation's *Steam*. The shift to online marketplaces has brought a lot of benefits for consumers: purchases are instant, fewer retail middlemen, software patches and updates are easy to install, and there is less physical media crowding living spaces. But there are also downsides. At the same time that software patches have become easier to deploy they have also come to be expected. Users have grown accustomed to software being buggy at launch.

At the time of writing (2018), *Star Citizen* (a crowd-funded space combat game) is a recent example of a controversial video game project[2]. But there are many examples of troubled projects in the game industry: *Aliens: Colonial Marines*, *No Man's Sky*, some entries in the popular *Assassin's Creed* series.

left-pad

Node (`nodejs.org`) [16] is an open source Javascript (JS) runtime for creating command line tools and network applications. Node was first developed in 2008 by Ryan Dahl of Joyent. It is based on the V8 JS runtime engine, developed

[2] https://motherboard.vice.com/en_us/article/ne5n7b/star-citizen-court-documents-reveal-the-messy-reality-of-crowdfunding-a-dollar200-million-game

by Google originally for the Chrome browser. Node enables the use of JS as a *server-side* language.

The default package management tool for Node is called `npm`. The company npm, Inc. maintains a large software registry at `npmjs.com`. Because of the widespread popularity of Node, many projects have a dependency in the `npm` registry. In 2016, as a result of a dispute over naming, a developer unpublished a package called *left-pad* from the `npm` registry. Many popular Node projects depended upon *left-pad* and its deletion caused the `npm` tool to become effectively unusable until the problem was sorted out. As a result of this incident, npm, Inc. changed its policies to restrict developers from removing their packages from npm.

Therac-25

Required Readings — Leveson and Turner [17]

Therac-25 [17] was a "dual mode" radiotherapy machine. The machine worked by accelerating electrons to high energies. The treatment area on the patient could either be exposed directly to the beam of electrons or the electrons could first be converted to X-rays (this is the meaning of "dual mode"). A special device (collimator) placed in the path of the electron beam accomplished the conversion to X-rays. The conversion resulted in a large energy loss which meant that the X-ray mode required a very high energy beam of electrons. The danger was that a patient might, accidentally, be exposed to the high energy electron beam if the machine was in X-ray mode but the collimator was not in place. Previous Therac models protected against this danger with hardware interlocks. However, in Therac-25, more of the safeties were handled purely in software. During the mid 1980s, software failures in Therac-25 caused at least 6 radiation overdose accidents.

Portions of the Therac-25 software were carried over from previous models. The software was written in PDP-11 assembly by a single developer. Lack of the hardware safety measures in previous models greatly increased the danger of software failures in Therac-25. Leveson *et al* [17] suggest that poor software engineering practices contributed to the Therac-25: insufficient documentation, quality assurance, logging, and testing.

1.2 Sources and Materials

In this section, we would like to provide a rough topical outline of the course and indicate sources from which the content was drawn. Many of these sources will be available through the library resources and will be assigned as readings. We will have approximately 28 primary lectures and 14 lab sessions over the course of the semester.

- **Object-Oriented Programming and Design**: In this section, we will explore object-oriented programming through the lens of *design patterns*. The seminal reference for design patterns is the so-called "gang-of-four" book [18]. Of course that text is a classic but we will also draw on *Head*

First Design Patterns [19] as it is explicitly Java-based and written in a very accessible style. In addition to design patterns we will cover the essential kinds of UML notation [20, 21]; GRASP patterns [1]; SOLID principles [22]; cohesion, coupling, and other code metrics [23, 24]. Your refresh of Java programming should cover the basic elements of object orientation such as *inheritance* and *encapsulation*.

General references for object-oriented design that can be found in the literature are the articles by Booch [25], Meyer [26], and Hatton [27].

- **Software Processes**: *Process* is a technical term in Software Engineering. A process describes *how* a development team produces software. Software life cycle processes are described in an IEEE Standard [28]. We will cover the *waterfall* model [29] and its activities. We will talk about what is meant by *iterative* and *incremental* development [30]. Following that we will talk about the various popular *agile* methodologies [31]: eXtreme Programming (XP) [8], Scrum [32], Kanban, and Lean [33]. Time permitting we will also discuss other topics such as the DevOps revolution.

- **Requirements Engineering**: *Requirements* describe how a system is supposed to behave. The gathering of requirements from stakeholders is one of the most important planning activities. We cover *use cases* [34] and *user stories* as a means of documenting requirements.

- **Testing**: Testing helps a developer estimate the correctness of software. Test-Driven Development (an agile practice) incorporates unit testing into development at a fundamental level. We will learn unit testing and TDD with JUnit. This is a popular unit testing framework for Java with good IDE support.

- **Tools**: Effective use of tools is an important part of software engineering. In this course we will use a variety of popular tools for software development. See Section 1.2.1. One thing to note here, do not view the fact that we use certain tools as a limitation. Git and github are widely used and are language agnostic. In the case of maven, although it is specifically a Java (or JVM) tool, much of the knowledge you gain about builds through learning maven will be transferable.

- **Web Application Development**: Application experiences are increasingly being delivered via the browser. For this reason, developers should have some understanding of how web applications function. Since the course is Java-based, we will look at the basic components of Java server-side technology (Java EE). Additionally, we will explore front-end issues, single page applications, and the role of popular Javascript libraries and frameworks.

1.2.1 Materials

In terms of the availability of tools, it is a good time to be a developer. There are an abundance of technologies, programming languages, and development environments to choose from (many are available for free). In our view, you should, at minimum, gain experience with a development environment, a build tool, and version control.

The recommended IDE for this course is the Java EE (Enterprise Edition) version of Eclipse (`http://www.eclipse.org`). This version of the IDE allows you to do everything that you could in Java SE but also integrates server-side features.

There are a number of build tools for Java: Ant, Maven, and Gradle. We will be using Maven in this course. Briefly, maven allows us to specify properties of a build in a special XML file, `pom.xml`. We will also cover the basics of version control with git and github.

- The Eclipse IDE is available at URL `http://www.eclipse.org/ide/`
- Maven is an apache project. Downloads and installation instructions can be found at `https://maven.apache.org/`
- Git is available from `https://git-scm.com/` and Github, for hosting remote repositories is at `https://github.com/`

1.3 Literature Milestones

This section gives a brief chronological overview of important works in the field of software engineering. Compared to some of the other sciences, software engineering is different in that books have played as important a role as the peer review literature in disseminating primary knowledge. So, our list of important works contains both papers and books.

- 1968, *go to statement considered harmful* E. Dijkstra [35]—This letter launched interest in structured programming. Also, it created a new genre of article in the computer science domain: the "considered harmful" letter.
- 1975, *The Mythical Man-Month* F. Brooks [36]—Classic collection of essays on software engineering. Essential reading for all developers.
- 1986, *Object-Oriented Development* G. Booch [25]—Booch presented one of the earliest, systematic, approaches to OO development. This article provides an example of what diagramming looked like prior to the UML.
- 1994, *Design Patterns: Elements of Reusable Object-Oriented Software* E. Gamma, R. Helm, J. Vlissides, and R. Johnson [18]—The seminal reference on design patterns in software, also called the "Gang of Four" book.

- 1997, *Death March* E. Yourdon [37]—Yourdon explores the forces and influences that lead to doomed software projects.
- 1999, *Extreme Programming Explained* K. Beck [8]—eXtreme Programming (XP) is one of the major agile processes. XP promoted and popularized practices such as test-driven development, pair programming, and refactoring.
- 2001, *Manifesto for Agile Software Development*[3]—The manifesto codified the values and principles of the agile movement.
- 2003, *Domain-Driven Design* E. Evans [38]—This is a classic text on modeling for software development. Introduced influential ideas such as the bounded context and the ubiquitous language.
- 2003, *Lean Software Development* T. and M. Poppendieck [39]—The Poppendiecks applied ideas from lean manufacturing to software development.
- 2011, *The Lean Startup* E. Ries [40]—In this influential book, Eric Ries applied lean thinking to the silicon valley startup.
- 2013, *The Phoenix Project* G. Spafford, K. Behr, and G. Kim—This is a novel about IT that makes the case for DevOps.

1.4 Exercises

1. Find an example of when a programmer (or computer scientist or IT specialist) has been portrayed in popular culture, e.g. in a movie or television series. The example I give in lecture is Dennis Nedry from the movie *Jurassic Park*.

   In a short essay (of, at least, 700 words), address all of the following points: what are the positive and negative aspects of the portrayal; was the subject a *professional* according to the ACM/IEEE Code of Ethics and Professional Conduct (provided in slide set for lecture 1); would you be comfortable being associated with the subject, why or why not.

2. As described in lectures, this course includes a semester team project. Please submit a list of your team members. The teams should have between 4 and 6 members (these limits will be strictly enforced). You should also come up with a unique name for your team.

[3] agilemanifesto.org

Chapter 2

Object-Oriented Development and Design Patterns

This chapter provides some important foundations for object-oriented development. We review the basic concepts of object-oriented programming. We introduce the Unified Modeling Language (UML) and UML class diagrams in particular. And we introduce the important idea of design patterns, which we build our presentation around.

2.1 Classes and Objects

Required Readings — Larman [1, Ch. 16, UML Class Diagrams]. Booch [25, Object-oriented Development]

Classes are the basic unit of object-oriented development. A class is like a blueprint for the creation of *objects*. An object is defined by three pieces of data: the class it belongs to (its *type*), a state that is captured in a set of attributes or fields, and a collection of methods that operate on the state of an object. The Java Tutorials OCA Prep [2] provides a perfectly reasonable introduction to object-orientation (in the Java context).

In software engineering, the exact meaning of certain terms, such as "Object-Oriented", can be difficult to pinpoint. In attempting to grasp this concept, it is important to distinguish between how object-orientation is implemented and what it is trying to achieve. Our focus is on the latter (the former belongs to the theory of programming languages and might begin with a discussion of topics like *dynamic dispatch* [41]). In a general sense, object-orientation is about mental models. Objects are supposed to be "like" the representations human beings use to understand the world. Therefore, we should find it easier to compose complex systems out of objects compared, for example, to functional decompositions [26]. This idea of convergence of conceptual categories and programming categories is also called *low representational gap* and you can find a discussion of it in the textbook [1, p. 138]. Consider the following quote from Booch.

> Abstraction and information hiding are actually quite natural activities. We employ abstraction daily and tend to develop models

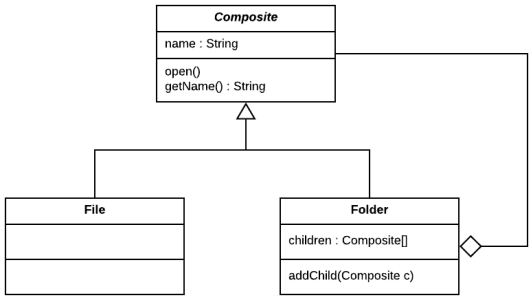

Figure 2.1: UML Class Diagram

> of reality by identifying the objects and operations that exist at each level of interaction. (Grady Booch, 1986 [25])

Object-oriented designs are often documented using a diagramming notation called the Unified Modeling Language (UML) [21, 20]. UML is a huge specification and it is doubtful that more than a few people understand all of it deeply. However, it is not too hard to grasp the basics and it is the standard for visual modeling in the industry. UML includes different types of diagrams to represent different aspects of a system: *static structure*, *behavior*, *communication* and so on.

Figure 2.1 shows an example of a UML class diagram. Each type (class or interface) is represented by a box with three sections: name of the type, fields, and methods. Arrows between the boxes indicate various kinds of relationships between the types (Gestwicki gives a readable summary [20]). Two of the most important kinds of relationships between classes are *inheritance* and *composition* (aggregation). Inheritance means that one class specializes another class. For example, a Cat *is an* Animal. Composition means that one class contains or uses another class. For example, a Car *has an* Engine. In UML class diagrams, relationships are represented by different types of lines and arrowheads. The text [1, p. 249] contains a reference chapter on UML class diagrams.

In object-oriented design, a common piece of advice is to "favor composition over inheritance" [18, 19]. What this means is that it is better to assemble a complex system by putting things together rather than trying to design a complicated hierarchy of classes. Inheritance is best used judiciously. Evidence shows, for example, that designs with deep inheritance trees are more error prone [23].

In an object-oriented context, encapsulation, delegation, inheritance, composition, etc. are the basic tools used to construct programs and systems. While it is essential to understand how these basic tools operate, a developer/designer needs to know more. By analogy, to be an artist, it is not enough to own a set of brushes and an easel. That naturally raises some questions: what

is already known, and what *expertise* is out there? The answer, in both cases, is *quite a lot*. People have been doing object-oriented development for around a half-century (in 1967, the programming language *Simula* introduced the notion of classes as a means of encapsulating data [25][1]). In all that time, software developers have discovered how to work successfully within the object-oriented paradigm. One way this knowledge has been recorded is in numerous *Design Patterns* [18, 19].

Patterns are an idea from architecture that was applied to software development[2]. We quote the definition of design patterns from the the "Gang of Four" book (GoF) [18], probably the most influential source on this topic.

> Description of communicating objects and classes that are customized to solve a general design problem in a particular context.
>
> (Gamma *et al* 1995, [18])

Larman [1] describes patterns as "a vocabulary of principles and idioms that can be combined to design objects".

Design patterns do not include an implementation. Nor are they simply templates. Patterns can be thought of as the time-tested solutions that have been *discovered* rather than *invented*. Patterns indicate how classes should be structured to solve particular problems and usually rely on polymorphism and delegation. There is also a large emphasis on designing for changes that can be made at run time rather than compile time. In Section 2.2, we look at a number of GoF patterns as well as some, such as *dependency injection*, that came after the GoF book.

2.2 Patterns

Required Readings — Larman [1, Ch. 17, GRASP: Designing Objects with Responsibilities]. Larman [1, Ch. 26, Applying GoF Design Patterns]

GoF [18] includes descriptions of 23 different object-oriented design patterns. Developers can use pattern names as a shorthand for communicating intricate properties of a system architecture. For that reason alone, it is helpful to learn the terminology associated with design patterns.

We do not intend to cover all of the GoF patterns in this section. We only want to cover enough to communicate the essence of the topic. Information about patterns is widely available online. Also, our treatment is heavily influenced by the presentation of the topic from the *Head First* series [19].

Patterns are categorized by the the type of problem they solve. The list below gives some of the common categories of patterns as well as lists the names of the particular patterns we will discuss.

[1] Moreover, in the 1980s and 1990s and largely due to the popularity of graphical-user interfaces, object-orientation became the dominant programming paradigm with languages such as *C++* and *Smalltalk*.

[2] For example, there are problems in architecture that are common to many buildings: how to create a hallway with good natural lighting, how to design a staircase etc. Over time, through experience and trial-and-error architects have found what works and what doesn't.

- Design patterns consist of a: *Name*, *Problem Description*, *Solution* (no implementation)
- Types of Patterns
  - *Creational*: Singleton, Dependency Injection, Builder
  - *Behavioral*: Strategy, Observer, Template Method
  - *Structural*: Composite, Decorator

2.2.1 Composite

The composite pattern is a way of building tree-like objects that can be reconfigured at run-time. Trees are familiar to Computer Scientists in many different guises. A file system is one example of a tree-like structure that is encountered in computing. In this case, the tree arises from the fact that folders can contain both files and other folders. In terms of the composite pattern, the property of "can be contained in a folder" would be abstracted into an interface or abstract class that both files and folders inherit from. Figure 2.1 shows a partial sketch of the class diagram for file-system example. The GoF definition of the composite pattern is,

> Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. (Gamma *et al* 1995 [18])

We will explore the composite pattern using the example of simple arithmetic expressions like $(1+3)*(5-2)$. These expressions have a natural tree-like structure related to their syntax. In evaluating $(1+3)*(5-2)$ we understand that we should first perform the addition and subtraction and then do the multiplication as the last step. Arbitrary expressions like this have the structure of a binary tree where the interior nodes correspond to arithmetic operations while the leaf nodes correspond to atomic values[3].

To apply the composite pattern to this problem we should map our basic types (operations and values) to the types shown in Fig. 2.1. Clearly, the atomic values correspond to `Leaf` while the various arithmetic operations correspond to `Composite`.

- Overall supertype: `Term`
- Atomic value type: `Vals`
- Operation type: `Opers`
  - Subtypes to represent different types of binary operators

[3]Multi-argument expressions like $(1+4+9+16)$ are also binary trees once we choose an order of operations, e.g. $(1+(4+(9+16)))$

A Java implementation is shown in Listing 2.1. Note that `Term` is the overall superclass (an *abstract class* in this case). The class `Opers` is also abstract. We can implement different operations by subclassing `Opers`.

Listing 2.1: Concrete Operation Example

```
abstract class Term {
  public abstract double getValue();
}

class Vals extends Term {
  private double myValue;

  public Vals(double m) {
    this.myValue = m;
  }

  public double getValue() {
    return myValue;
  }
}

abstract class Opers extends Term {
  protected Term leftVal;
  protected Term rightVal;

  public Opers(Term l, Term r) {
    this.leftVal = l;
    this.rightVal = r;
  }
}
```

There is only one method shown in the implementation which is called `getValue()`. The purpose of this method is to compute the actual *value* of a term (i.e. to evaluate it). A `Vals` simply evaluates to the value contained in `myValue`. Calling `getValue()` on a composite class will start a recursive computation of the value of the term. A concrete operation `Adding` is shown in Listing 2.2.

Listing 2.2: Composite Pattern Code

```
public class Adding extends Opers {
  public Adding(Term l, Term r) {
    super(l, r);
  }

  @Override
  public double getValue() {
    return this.leftVal.getValue() + this.rightVal.getValue();
  }
}
```

Of course, we could implement more methods to give us greater control over the `Term` objects at runtime. For example, by adding setter methods for the `l` and `r` instance variables we could easily reconfigure a `Term` at run-time or partially evaluate it.

We can use the design by nesting constructors of the `Vals` and `Opers` classes. For example, the `x` in Listing 2.3 is the expression $(2 + 2) * (2 + (2 * 2))$.

Listing 2.3: Building Arithmetic Expressions

```
Multiplying x = new Multiplying(
  new Adding(new Vals(2.0), new Vals(2.0)),
    new Adding(new Vals(2.0),
      new Multiplying(new Vals(2.0), new Vals(2.0))
    )
);
```

2.2.2 Observer

The *Observer* pattern answers a question we commonly encounter in software development: how to coordinate communication (method calls, variable assignments, etc.) when objects in one part of an application need to know about changes to objects in a different part of the application. For example, suppose we are designing the user interface of some application, there might be an object that represents UI elements (buttons, text fields) and we would like other components in our application to respond or to be notified when the user interacts with the UI (e.g. clicking on a button, pressing a key, etc.). Generally, the observer pattern is useful when our system needs to deal with events.

By employing the features of object-oriented languages, there is an elegant solution. There is more than one set of terms associated with the Observer pattern: observable/observer, publisher/subscriber, and subject/listener.

The description of the observer pattern from GoF is as follows.

> Define a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically. (Gamma *et al* 1995 [18])

The observer pattern sets up a publisher/subscriber relationship between objects. Subscriber objects can register with a publisher and will receive messages when the publisher pushes updates. Moreover, publishers and subscribers should only know about each through well-defined and concise interfaces. This is an example of the more general principle of *low coupling*, which is discussed in Section 2.3.

In our example application, the *publisher* will be a sensor (for example, this might be a thermometer) and the *subscribers* will be called listeners. The listeners need to keep up-to-date on the events generated by the sensor. To realize the pattern we need to create interfaces for the subject and listeners, and then we need to create concrete classes that implement those interfaces (see the class diagram).

The interface types for the Observer example are shown in Listing 2.4. These interfaces represent the contracts that the sensors and listeners have agreed upon. To proceed, we have to provide concrete implementations of these interfaces.

Listing 2.4: Observer: Interfaces

```
interface Sensable {
    public void addListener(Listens l);
    public void removeListener(Listens l);
```

```
    public void notifyListeners();
}

interface Listens {
    public void update();
}
```

Listing 2.5 shows the concrete implementation of the sensor. Note that the `SensorSubject` class has one private field which is a `Set` (with an implementation from the Java collections). This collection is how the sensor keeps track of the objects that are listening to it. We use a `Set` because we want to make sure that we don't add an object twice. Note that we only have one method that is not inherited from the `Sensable` interface. The `eventHappened()` method is a way for us to simulate a sensor event.

Listing 2.5: Observer: Concrete Sensor

```
class SensorSubject implements Sensable {
    private Set<Listens> sensorListeners = new HashSet<Listens>();

    @Override
    public void addListener(Listens l) {
        sensorListeners.add(l);
    }

    @Override
    public void removeListener(Listens l) {
        sensorListeners.remove(l);
    }

    @Override
    public void notifyListeners() {
        for(Listens e : sensorListeners) {
            e.update();
        }
    }

    public void eventHappened() {
        this.notifyListeners();
    }
}
```

Listing 2.6 shows a concrete `Listener`. The `update()` method says what the listener should do when a change occurs (in this case, it prints to standard output). We also provide a constructor so that each object can be given a name.

Listing 2.6: Observer: Concrete Listener

```
class SensorListener implements Listens {
    private String name;

    public SensorListener(String name) {
        this.name = name;
    }

    @Override
    public void update() {
        System.out.println("Message Received by " + this.name);
    }
}
```

To actually see our implementation of the observer pattern in action we just have to wire everything together. One way to do this is shown in Listing 2.7.

Listing 2.7: Observer: Main

```
public class ObserverExample {
    public static void main(String[] args) {
        SensorListener o1 = new SensorListener("Listener 1");
        SensorListener o2 = new SensorListener("Listener 2");
        SensorListener o3 = new SensorListener("Listener 3");

        SensorSubject sensor = new SensorSubject();

        sensor.addListener(o1);
        sensor.addListener(o2);
        sensor.addListener(o3);

        // simulate sensor event
        sensor.eventHappened();
    }
}
```

The last statement of the `main` method (the call to `eventHappened()`) cues the sensor to publish an update to all of its subscribers. In the console, we will see the output from each of the listeners' `update()` methods. The textbook [1, p. 463] also discusses the Observer pattern and gives UML Sequence diagrams that illustrate the behavior of the pattern.

2.2.3 Strategy

The *Strategy* pattern is about being able to reconfigure behavior at runtime. The pattern uses encapsulation and delegation to make algorithms interchangeable. The definition from GoF is as follows,

> Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. (Gamma *et al* 1995 [18])

In GoF, Strategy is motivated by the example of adding line breaks in text. Consider the class `MyText` in Listing 2.8. The `buff` field stores a `String`, which can be arbitrarily long. We might want to display `buff` in a particular area of the screen. The screen area will have finite width and therefore the text will need to wrap and we will need to add line-breaks. However, there are a variety of ways we could add line-breaks depending on the width and the desired visual effect. For instance, the text can be centered or justified. Long words can be broken up to achieve better overall spacing.

The way `MyText` is written in Listing 2.8 is fine if we only want to have one style of line-breaking. However, if we wanted to have different styles of line-breaking then we would be forced to rewrite the `makeLineBreaks()` method. Moreover, the design in Listing 2.8 does not allow for the line-break style to change at runtime. By using the strategy pattern we can achieve a more flexible design.

Listing 2.8: Text Buffer Class

```
public class MyText {
  private String buff;

  public MyText(String buff) {
    this.buff = buff;
  }

  public String makeLineBreaks() {
    return // Implement linebreaking here
  }
}
```

In the Strategy pattern, an abstraction encompasses the behavior that will vary (line-breaking algorithm in our example). Listing 2.9 shows how things could be set up. The interface `BreakBehavior` encapsulates the algorithm. Note how the line-breaking behavior is delegated to the field `b`.

Listing 2.9: Line Break Strategy Example

```
interface BreakBehavior {
  public String linebreak(String s);
}

class MyText {
  private String buff;
  private BreakBehavior b;

  public MyText(String buff, BreakBehavior b) {
    this.buff = buff;
    this.b = b;
  }

  public String makeLineBreaks() {
    return b.linebreak(buff);
  }

  public void setB(BreakBehavior b) {
    this.b = b;
  }
}
```

Concrete line-break behavior classes are shown in Listing 2.10. Note that we are using the `WordUtils` class from Apache Commons[4]. Chapter 4 covers tools you can use to easily include third-party libraries in your projects.

Listing 2.10: Line Break Strategy Example

```
import org.apache.commons.text.WordUtils;

class SimpleBreakBehavior implements BreakBehavior {
    @Override
    public String linebreak(String s) {
        return WordUtils.wrap(s, 60);
    }
}

class NoBreakBehavior implements BreakBehavior {
    @Override
```

[4]Apache Commons is a project of the Apache Software Foundation. It has a large number of reusable Java components and libraries applicable to all types of situations and provided with a permissive open-source license.

```
    public String linebreak(String s) {
        return s;
    }
}
```

Finally, Listing 2.11 shows how we can actually use this class and vary the behavior at runtime. Note that `s` is a long single-line string. To change how `MyText` breaks lines we set the `BreakBehavior` field.

Listing 2.11: Line Break Strategy Example

```
public class StrategyExample {
        public static void main(String[] args) {
                String s = "The Sages say truly " +
                                "That two animals are in this forest: " +
                                "One glorious, beautiful, and swift, " +
                                "A great and strong deer; " +
                                "The other an unicorn.";

                MyText text = new MyText(s, new SimpleBreakBehavior());
                System.out.println(text.makeLineBreaks());

                System.out.println("\n~~~~~~ Changing Line Break Behavior\n");
                text.setB(new NoBreakBehavior());
                System.out.println(text.makeLineBreaks());
        }
}
```

The textbook [1, p. 447] presents the Strategy pattern in the context of the running examples and also shows some relevant UML diagram.

2.2.4 Singleton

Singleton [1, p. 442] is probably the most controversial of the GoF design patterns. However, its usage is widespread and it is part of the architecture of many frameworks (in UML class diagrams there is even a special notation to show that a class is a singleton). The pattern essentially allows developers to create a global object. In that sense, singleton has all the issues of global variables and must be treated carefully, especially in a concurrent setting. The GoF define the Singleton pattern as follows.

> Ensure a class has only one instance, and provide a global point of access to it. (Gamma *et al* 1995 [18])

The point to remember with the Singleton pattern is that we are creating a unique object. We are not just storing some data in a static context. In Java, a Singleton can be implemented as follows,

- Mark the constructor `private`: We usually do not pay too much attention to the visibility of constructors. Marking the constructor `private` ensures that `new Singleton()` can only be called within the class itself. This gives us control over the creation of instances.
- Create a `static` field of type `Singleton` on the `Singleton` class: This field will reference the unique instance. Although it seems a bit self-referential, this is the standard way to implement a singleton.

- Write a getter method to retrieve the Singleton instance: the getter checks if the unique instance is `null` if so it is instantiated and returned else the current instance is returned

The complete implementation of the Singleton is shown in Listing 2.12.

Listing 2.12: Complete Singleton Example

```
public class Singleton {
   private static Singleton uniqueInstance;

   private Singleton() {}

   public static Singleton getInstance() {
      if (uniqueInstance == null) {
         uniqueInstance = new Singleton();
         }

      return uniqueInstance;
      }
}
```

As Larman [1, p. 445] explains, a common question about the Singleton pattern is why an object is preferred to some static methods and variables. His justifications are that static methods are not polymorphic, and also that a Singleton will be easier to extend to a "regular" class if non-unique instances are needed in the future.

2.2.5 Decorator

Anyone who has spent substantial time programming in Java has likely encountered the *Decorator* pattern. The syntax of reading files and creating streams in Java is based on the Decorator pattern. GoF suggest that Decorator is about extending the behavior of a class without inheritance.

> Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. (Gamma *et al* 1995 [18])

In Java, when we want to open a file we often use a syntax like that shown in Listing 2.13.

Listing 2.13: Example: Streams and File Readers

```
BufferedReader br = new BufferedReader(new FileReader("./text3.txt"));
```

We begin with an anonymous `FileReader` that is passed to the constructor of a `BufferedReader`. The `BufferedReader` is said to *wrap* or to *decorate* the `FileReader`. But what is going on here exactly? The `BufferedReader` adds a new functionality (namely, a buffer) to the `FileReader`. The file could have been read with a plain `FileReader` but maybe performance demanded a buffer. Instead of creating many individual of subclasses of `FileReader` to handle all of the combinations of options that someone might want to put on a reader,

the designers of the `java.io` libraries decided to use the Decorator pattern. The options are reified in the class hierarchy.

The Decorator pattern helps in keeping the number of classes manageable. The example commonly discussed in the context of the Decorator pattern is the windowed user interface [19]. In such a UI system, there might be a base class that represents a basic window (a box with some text inside). The system should also allow for many variants of the basic window depending on the situation: scroll bars, menus, or fancy graphical borders. Instead of creating a new subclass for each possible window variation (e.g. basic window + scroll bar, basic window + menu, basic window + scroll bar + menu) the decorator pattern allows us to create one class for each option.

We will explain the Decorator class with an example. The example we look at is about styling and displaying text. The base class has a way to store some text (in a `String`) and display it to the console. The decorator classes style the text in different ways. This is only meant as an example to show how the pattern is organized.

Listing 2.14 shows the abstractions for our Decorator example. The base classes and the decorator classes will all inherit from `TextComponent` (this is the overall superclass). All of the decorator classes will inherit from `TextDecorator`.

Listing 2.14: Example: Streams and File Readers

```
abstract class TextComponent {
  abstract public String produceText();
}

abstract class TextDecorator extends TextComponent {
  protected TextComponent next;

  public TextDecorator(TextComponent t) {
    this.next = t;
  }

  public String produceText() {
    return this.next.produceText();
  }
}
```

The base class and a concrete decorator class are shown in Listing 2.15. The base class has a field to store a `String` and a method to print it to the console. The class `DashBorderDecorator` can wrap the base class and also add a border of dashes to the text.

Listing 2.15: Base Class and Dash Border Decorator

```
class TextBase extends TextComponent {
  private String s;

  public TextBase(String s) {
    this.s = s;
  }

  @Override
  public String produceText() {
    return s;
  }
}
```

```
class DashBorderDecorator extends TextDecorator {
  public DashBorderDecorator(TextComponent t) {
    super(t);
  }

  @Override
  public String produceText() {
    return "--- " + super.produceText() + " ---";
  }
}
```

Listing 2.16 shows a class with a `main` method that illustrates how the pattern works. The console output is shown in Listing 2.17. Notice that we are using Java *reflection*[5] to show exactly the runtime type of `tx` each time it is redefined. Also, note that since we call `new` 3 times this method creates 3 different objects in memory.

Listing 2.16: Main Method for Decorator Example

```
public class DecoratorExample {
  public static void main(String[] args) {
    TextComponent tx = new TextBase("The Text!");
    System.out.println(tx.getClass().toString());
    System.out.println(tx.produceText());

    tx = new DashBorderDecorator(tx);
    System.out.println(tx.getClass().toString());
    System.out.println(tx.produceText());

    tx = new CapitalDecorator(tx);
    System.out.println(tx.getClass().toString());
    System.out.println(tx.produceText());
  }
}
```

Listing 2.17: Console Output for Decorator Example

```
class kz.edu.nu.cs.teaching.TextBase
The Text!
class kz.edu.nu.cs.teaching.DashBorderDecorator
--- The Text! ---
class kz.edu.nu.cs.teaching.CapitalDecorator
--- THE TEXT! ---
```

2.2.6 Dependency Injection

Dependency injection is a topic that seems simple at first and one can wonder if a separate term is really needed to describe the concept. However, as a software engineer, it is important that you understand what it means to people in industry and make an attempt to understand it deeply. The concept is present in many standards and frameworks. For example, in Java, there is an annotation `Inject` that lets a developer describe dependencies. The concept is also central to frameworks such as Google Guice and Spring.

Dependency injection is about how objects should receive their dependencies. There is a saying in object-oriented development that entities should

[5]Reflection refers to the ability of a programming language to inspect itself.

"depend on abstractions, not concretions". One way to try to adhere to this principle is to eliminate the keyword `new` whereever possible[6]. In his agile programming book Robert Martin [22] elaborates further. Consider the following quote,

> - No variable should hold a reference to a concrete class
> - No class should derive from a concrete class
> - No method should override an implemented method of any of its base classes
>
> (Martin *et al* 2006 [22])

Now let us describe the dependency injection pattern with some code examples. First of all, what is a *dependency*? A dependency is simply some other class that is needed by the class under consideration. Oftentimes, the dependencies are the external references that an object holds to other classes: instance variables and variables received as parameters. Please refer to the discussion of UML class diagrams [1, p. 260] to review some of the ways classes can be related to each other.

Consider Listings (Java) 2.18 and 2.19, they both show the definition of a simple class (`myclass`) where the dependencies are resolved either with dependency injection or without dependency injection. In Listing 2.19, the class receives references for the `a` and `b` fields through the constructor (specifically, this is called *constructor injection*). Note that the `new` keyword does not appear in Listing 2.19. In Listing 2.18, the class is itself responsible for instantiating `a` and `b`. In 2.18, note the distinction between the abstract and concrete classes `DependencyA` and `RealDependencyA`. The code strongly suggests that `DependencyA` is an *abstract* type (an abstract class or an interface) while `RealDependencyA` must be a concrete class.

Listing 2.18: Resolving without Dependency Injection

```
public class MyClass {
   DependencyA a = new RealDependencyA();
   DependencyB b = new RealDependencyB();
}
```

Listing 2.19: Resolving with Dependency Injection

```
public class MyClass {
   DependencyA a;
   DependencyB b;

   public MyClass(DependencyA a, DependencyB b) {
      this.a = a;
      this.b = b;
   }
}
```

[6] Obviously, we cannot entirely remove `new` from an application. In practice, objects must be instantiated at some point. But we would like to, somehow, isolate object creation from our business logic.

What is the difference between the cases (and why should we care)? To make sense of this, we have to imagine that `MyClass` is part of some larger system. Of course, in a simple example, it might not make much of a difference which style of dependency resolution is used. However, suppose there are many possible types that inherit from `DependencyA` and `DependencyB`. In Listing 2.19, these classes could be interchanged very easily since there are no hard dependencies on the concrete classes. Only the constructors that appear elsewhere have to be changed. On the other hand, to change the concrete classes in Listing 2.18 we have to actually edit `MyClass`.

Another advantage of dependency injection is that dependencies are exposed through the constructor (with the constructor becoming a kind of basic documentation). Notice that there is no constructor defined in Listing 2.18. To get an instance of `MyClass` we would write something like `MyClass obj = new MyClass();`. We would not necessarily be aware that the class instantiated `RealDependencyA` and `RealDependencyB`. With dependency injection, it is necessary for us to understand the dependencies in `MyClass` since we have to give them as constructor parameters. This also makes it possible for code hinting facilities of an IDE to provide more immediate feedback.

2.2.7 Template Method

According to Larman [1, p. 630], the *Template Method* pattern is "at the heart of framework design". The GoF definition of the pattern is as follows,

> Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. (Gamma *et al* 1995 [18])

GoF motivates Template Method with the example of a document editing framework. Freeman *et al* [19] use an example of preparing different kinds of beverages beverages that require brewing. We will briefly recount the latter example. Tea and coffee are similar in that both are brewed, both contain caffeine (usually), and both are prepared using a similar sequence of steps: boil water, brew, and add condiments. Since the process (that is, the *algorithm*) for preparing these beverages is similar we should be able to define a common abstraction. Some of the steps will be identical such as boiling the water but some will be different. For example, we do not add all the same condiments to tea and coffee.

The idea of the Template Method pattern is that the abstraction defines some *primitive* operations that define the steps of some algorithm. A *hook operation* provides default behavior that subclasses can extend and "hook" into the algorithm at a certain point. All of the operations are called in sequence in the *template method*.

Android development is an example of template method.

2.2.8 Builder

In object-oriented programming there is often a need to create complex objects. For example, we may need to set many fields and guarantee that an object is not created in an incomplete state. The *Builder* pattern concerns the construction of complex objects. The GoF description is as follows,

> Separate the construction of a complex object from its representation so that the same construction process can create different representations. (Gamma *et al* 1995 [18])

The Builder pattern involves the coordination of four types.

- Builder: interface for building the parts of an object
- ConcreteBuilder: implements the Builder interface
- Director: constructs an object using a Builder
- Product: domain object that is being constructed

A *fluent* interface is often used with the Builder pattern. In a fluent interface the methods are called in a single large statement via method chaining. In the Builder pattern, at the end of the method chain, the `build()` method is called to actually return the object.

For the example, we will imagine that we are working on a reservation management system for an airline. As part of this work a class needs to be written to model tickets. The `Ticket` objects will be instantiated with a builder. A ticket has various information that needs to be collected: origin and destination of the flight, the date of the flight, and the desired cabin class (for example, economy, economy plus, first class).

Listing 2.20 shows how we would call the Builder for the airline example and how the fluent interface would look. This shows how a `Ticket` is constructed from a `TicketBuilder`. Notice that we have attempted to name the methods of the Builder class so that they correspond to natural language.

Listing 2.20: Builder with Fluent Interface

```
public class App {
  public static void main(String[] args) {
    TicketBuilder tb = new TicketBuilder();

    Ticket t = tb.from("Astana")
                 .on("March 12")
                 .to("Frankfurt")
                 .withClass("Economy")
                 .build();

    System.out.println(t);
  }
}
```

Listing 2.21 shows the `Ticket` class. Notice that we have defined a constructor that takes a Builder. The way that `this` is used in the constructor is called *constructor chaining*.

Listing 2.21: Builder with Fluent Interface

```
public class Ticket {
  private String origin;
  private String destination;
  private String date;
  private String travelClass;

  public Ticket(String orig,
    String dest,
    String date,
    String trav) {
      this.origin = orig;
      this.destination = dest;
      this.date = date;
      this.travelClass = trav;
    }

    public Ticket(TicketBuilder tb) {
      this(tb.origin,
           tb.destination,
           tb.date,
           tb.travelClass);
      }
}
```

Listing 2.22 show the `TicketBuilder` class. Notice that the fields of `TicketBuilder` are the same as `Ticket` class. Also note that the return types of all of the builder methods return a `TicketBuilder` and that the actual object returned is `this`.

Listing 2.22: Builder with Fluent Interface

```
public class TicketBuilder {
  String origin;
  String destination;
  String date;
  String travelClass;

  public TicketBuilder from(String f) {
    this.origin = f;
    return this;
  }

  public TicketBuilder to(String t) {
    this.destination = t;
    return this;
  }

  // Other builder methods

  public Ticket build() {
    return new Ticket(this);
  }
}
```

Some things to note about the Builder pattern. One reason to use the pattern is to make sure that an object cannot be created in an invalid state. For example, we do not want to have a `Ticket` with null fields or routes that

do not exist. The Builder pattern allows us to keep the logic for creation out of the `Ticket` class.

2.2.9 Exercises

1. Composite Pattern–Complete the arithmetic expression composite pattern as described above and add implementations for the four basic arithmetic operations: addition, subtraction, multiplication, and division

2. Composite Pattern–Create necessary methods to print terms in a visually appealing way

3. Composite Pattern–Add a method `getDepth()` that will compute the depth of the tree below a given term (assume a `Vals` has a depth of 0)

2.3 Cohesion and Coupling

In the previous section we introduced a series of design patterns. Now we would like to talk about some general design principles that these patterns embody. In fact, many different authors have attempted to produce a list of principles that captures the essence of good object-oriented design. Some of these lists have even been given fancy mnemonics.

The key goal that underlies the GoF patterns is *designing for change* (this change might include the possibility of reusing the code in the future). Most systems will need to respond to change at some point during their lifetime. One way to create a system where changes can be made easily is to "encapsulate what varies." For example, in the strategy pattern, the possibilty that new behaviors will be needed in the future is anticipated. Therefore, the particular behavior we are looking at is abstracted behind a well defined contract and the client only holds a reference to the interface. Freeman *et al* [19] collect and condense many of the best practices from the GoF book.

In the software engineering literature, common guidance is that systems should have *low coupling* and *high cohesion*. You can find this advice enumerated in the GRASP patterns [1, p. 271]. You can also find it in the Design Patterns literature [19].

Cohesion indicates the strength of a class, where a strong class is one that fulfills a clear and singular mission. A class *lacks* cohesion when it brings together lots of unrelated functionality. This same idea is also expressed in the *single responsibility principle* from SOLID [22].

Coupling tells us how much one class needs to know about another. Tightly coupled classes know a lot about the inner workings of one another. The problem with high coupling is that changes in one part of a system are more likely to affect (break) classes and functions in other parts of a system. This makes the whole system brittle or resistant to change.

Imagine a software system designed with the principle of *low coupling* and *high cohesion* versus a system designed without that principle. Compared to

the latter, the former system will tend to have *more* classes and *smaller* classes. The idea *low coupling* and *high cohesion* can also be expressed by analogy with load bearing structures such as a truss bridge.

2.4 Code Metrics

Code Metrics (also *Software Metrics*) measure things about software. As we will see, the most crucial questions of software engineering concern human behavior: how people communicate, how to lead teams, and so on. But in any scientific or engineering endeavor, we need to be able to quantify things. For example, we want to be able to quantify some aspect of our code and see if it correlates with good or bad design.

One basic metric that most people are familiar with is the number of lines of code, which is abbreviated as LOC or kLOC (1000 lines of code). SLOC stands for *source lines of code*. There are, however, many different software metrics that quantify different aspects of the code. Some of these metrics, such as *cyclomatic complexity* can be applied in any generic programming environment (i.e. in a language where we have branching or typical control constructs such as conditionals and loops). Other metrics are specific to object-oriented programming languages.

In this section we will review and define various software metrics and discuss how to interpret them.

2.4.1 Cyclomatic Complexity

Required Readings — McCabe [24]

The *cyclomatic complexity* (CC) was defined by McCabe [24] based upon ideas from graph theory. To understand what CC is we first need to introduce the idea of the program control flow graph. When first learning to program, students are introduced to basic constructs like loops and conditional statements (`if`, `for`, and `while`). A program can execute differently depending upon the inputs that are provided or the state of the computer. For example, consider the simple Javascript function in Listing 2.23[7].

Listing 2.23: Simple Javascript Function

```
function myfunction (x) {
  if (x > 0) {
    // Branch A
    console.log("A");
  } else {
    // Branch B
    console.log("B");
  }
}
```

Depending on the value passed to `myfunction` either branch A or branch B runs. Obviously, each time the function is called only one of the branches will execute. And in general, when we run our programs we do not expect execution

[7] You can experiment with cyclomatic complexity at the website `jshint.com`.

to follow every possible branch. The number of different ways a program can execute will grow exponentially with the number of branches.

We can visualize the execution of a program as a path through a certain directed graph called the program *control flow graph*. Each node in the graph represents a sequential block of code. Each arrow out of a node represents a different branch that the program execution could take at that point (a node with multiple outgoing arrows is a decision point). A simple `if` statement, for example, is represented by a node with two outgoing arrows (see figure).

The CC is calculated from the control flow graph. Let E be the number of edges (arrows) and N the number of nodes in the graph. Then the CC is defined as follows

$$CC = E - N + 2 \tag{2.1}$$

This formula is a special case of the *Euler Characteristic*, which is a topological invariant of mathematical spaces. What this means, in our case, is that programs with the same amount/kind of branching will have the same CC regardless of how we draw the control flow graph. For example, consider simple programs that only consist of sequences of statements without any branches. If there are 3 blocks in the program then $N = 3$ and $E = 2$. The CC is $2 - 3 + 2 = \mathbf{1}$. On the other hand, if there are 9 blocks then $N = 9$ and $E = 8$. Equation 2.1 still gives an answer of **1**. What this tells us is that all programs that only consist of a sequence of statements without branching have the same CC. More generally, we can collapse any linear chain of sequential blocks in the control flow graph without changing the CC.

McCabe showed that the CC can be interpreted (or calculated) in a few different ways. Under reasonable assumptions, the CC of a program is equal to one plus the number of conditional statements (`if`, `for`, and `while`) in the program. Also, if the control flow graph is *planar* (can be drawn on a plane without any of the edges overlapping) then the CC is equal to the number of disconnected regions the plane is divided into by the control flow graph.

The CC has many important implications. Human beings can only attend to a finite number of things at any one time. A high CC implies that the program contains many (perhaps nested) conditional statements. Developers are more likely to make mistakes if they have to keep too many possible execution paths or corner-cases in mind. CC also has important consequences for testing (we discuss testing later).

2.4.2 Object-Oriented Metrics

Required Readings — Basili *et al* [23]

Our primary reference for Object-Oriented software metrics is the paper by Basili *et al* [23]. This reference collected many of the object-oriented software metrics that had been defined at that time and performed a number of experiments with them in a real development context. The metrics were originally defined by Chidamber and Kemerer [42]. Three of the metrics from Basili *et al* [23] are noted here: Depth of Inheritance Tree (DIT), Coupling Between

Object Classes (CBO), and Lack of Cohesion on Methods (LCOM). CBO and LCOM are attempts to quantify the *coupling* and *cohesion* properties from Section 2.3.

The idea of object-oriented software metrics is that they are (or should be) indicative of good or bad class design. For example, we have already encountered the idea that systems should exhibit *low coupling* and *high cohesion*. Then, as a rule of thumb, we can say that both CBO and LCOM should be small in well-designed systems. In this section, we review specific object-oriented metrics and provide examples showing how they are calculated.

In an object-oriented programming language with subtyping, classes are arranged in a hierarchy. The DIT is the maximum depth of the class hierarchy relative to each class. Basili *et al* [23] advise that well designed classes will look like "forests" as opposed to one giant tree.

Basili *et al* [23] provide the following definition of Coupling Between Object Classes.

> A class is coupled to another if it uses its member functions and/or instance variables. CBO provides the number of classes to which a given class is coupled. (Basili *et al* 1996 [23])

The definition of Lack of Cohesion is as follows.

> The number of pairs of member functions without shared instance variables, minus the number of pairs of member functions with shared instance variables. However, the metric is set to zero whenever the above subtraction is negative. (Basili *et al* 1996 [23])

Note that the number of pairs of methods is a binomial coefficient $\binom{N}{2}$ where N is the number of methods.

For example, if a class has 5 methods then there are 10 pairs of methods. Consider the `Person` class in Listing 2.24. There are 3 methods. Applying the binomial coefficient formula there are 3 pairs of methods (technically, there is also the constructor, but we will not consider it in this calculation). For each pair we consider whether methods reference a common instance variable.

- `getName()` and `getAge()`: do not access any of the same instance variables
- `getAge()` and `getDescription()`: do not access any of the same instance variables
- `getName()` and `getDescription()`: both depend on `name`

In total there are 2 pairs without shared instance variables and 1 pair with shared instance variables. For the `Person` class LCOM is equal to $2 - 1 = \mathbf{1}$

Listing 2.24: Example Class

```
public class Person {
  String name;
  String address;
  int age;

  public String getName() { return name.toUpperCase(); }
  public int getAge() { return age; }
  public String getDescription() {
    return getName() + " " + address;
  }
}
```

2.5 System Design

Required Readings — Larman [1, Part III]. Larman [1, Chs. 25-26]. Booch [25, Object-oriented Development]. Meyer [26]. Hatton [27]

The text [1] thoroughly covers system design with objects. This is not surprising as it is, primarily, an OOA/D book. So, in this section of the notes, the goal is to highlight critical readings on design from our textbook. Also, we have collected some relevant readings from scholarly and trade publications.

One of the earliest coherent methods of object-oriented design is the 1986 paper by Booch [25]. The key idea is to focus on "the objects that exist in our model of reality" [25]. Meyer also outlines an approach to object-oriented design [26]. Although OO programming has proven enormously popular it has also attracted criticism. One example of such criticism is the article by Hatton [27].

To get a good sense of object-oriented design you should read Part III (Chapters 8-22), Chapters 25-26 (GoF patterns) of the text [1].

Also, it is helpful to be mindful of the principles of good object-oriented design from the GoF Design Patterns book [18]. We recall their apophthegms in the following list.

- Program to an interface, not an implementation.
- Favor object composition over class inheritance.
- Encapsulate the concept that varies.

Chapter 3

Software Processes

This chapter covers software processes. We begin with the software development process model and introduce the major ideas: the classical *waterfall* approach, iterative and incremental development, and *agile* methods. For the project component of this course, we primarily follow *Scrum* but attempt to pull ideas from other agile methodologies such as XP and Lean. As the most popular agile process, it is useful to learn what Scrum entails. The textbook [1] uses the *Unified Process*. However, as the author states [1, p. 19], UP is meant as an example. The different agile software development processes share basic principles.

3.1 Waterfall and Agile

Required Readings — Larman [1, Ch. 2, Iterative, Evolutionary, and Agile].

A *process* is how you do something. Most of the complex goals that we would like to achieve, either in work or life, can be decomposed into collections of smaller tasks. In software engineering, such smaller tasks are usually called *activities*. A software development process determines how the activities are organized towards creating the finished product (i.e. working software) [28]. The development activities are things like *planning*, *design*, *implementation*, *testing*, and *maintenance*. Consider the following quote,

> The IEEE defines a process as "a sequence of steps performed for a given purpose" [IEEE-STD-610] (CMM [43])

In casual speech, the distinction between the terms computer programming, computer science, and software engineering is not always apparent. Obviously, both computer scientists and software engineers have significant responsibilities that do not involve writing code. It is easy to become biased towards the belief that implementation (coding) is the single most important activity within the software development process. This is not unexpected. Learning to program is a significant personal achievement and anyone wants to believe that their efforts went to a meaningful aim. On the other hand, a more mature outlook, free of the need to impress, encompasses the broad range of development activities

and recognizes the distinct value added in each step. In fact, one of the main principles of Lean software development (discussed later) is to *optimize the whole* [39, 33].

Whether or not they realize it, everyone who codes, by definition, follows a process. Absent any methodological discipline, a developer will pursue the most immediate problems, or the low-hanging fruit, or simply whatever habits dictate. If we do not hold ourselves accountable to good project/time management practices then we will just do what we do by default. However, the "default process" will not always be the most skillful approach. The basic default process comes down to trial-and-error: search problem keywords online, write some code and try to compile it, debug any errors that occur, and repeat until done. See [6, Section 4.1.2] for a discussion of the simplest process model. A default process may work for a while, but it will be unmanageable as the system grows in complexity or when additional developers need to be brought onboard.

Waterfall

One of the earliest published software processes is *waterfall* [29][1]. This process is represented in the Figure. In waterfall, development consists of a strict linear sequence of distinct activities. The output of one activity serves as the input to the next. This exchange of deliverables gives the waterfall process its name. For example, the output of the planning activity might be a document gathering all of the system requirements together (a software requirements specification). The design team will then take the requirements specification and produce a design document (capturing the design of the system in various UML diagrams). Obviously, the waterfall process will produce lots of documentation. For that reason, waterfall is also sometimes called a *document-driven* approach.

There are well-known disadvantages to the waterfall approach. The main issue is inflexibility. Waterfall assumes that the project scope and requirements are reasonably stable and knowable in advance. Waterfall fosters a "Big Design Up Front" mentality: all the important decisions are made at the beginning of the project when the problems are least well understood. Waterfall also tends towards organizational silos (as in grain silos). A siloed organization is one in which teams work on mutually exclusive areas of responsibility and inter-team cooperation is limited (for example, limited the various handoffs that occur between phases in waterfall).

In practice, projects need to manage change. For example, some critical third-party library or service may become deprecated or markets may shift unexpectedly. It is widely believed. Larman [1, p. 23] reports on some of the evidence-based studies find that the waterfall approach is associated with project failure.

[1] Note that this is the traditional reference for the waterfall process. However, upon careful reading, Royce actually advocated for an iterative approach.

However, the waterfall approach can be appropriate in certain situations, such as when there is a reasonable expectation that the requirements and scope are stable.

Iterative, Incremental, and Evolutionary Development

In this section, we present properties of software development processes that contrast with the waterfall approach described in Section 3.1. Iterative, incremental, and evolutionary development (I will say *iterative and incremental*) are not new ideas. Larman and Basili [30] review the history of these ideas back to the early days of computing.

- *Iteration*: A single pass through the waterfall process
- *Incremental*: Delivering additional features in small units
- *Evolutionary Model*: Iterative model where the software gradually evolves over time, allows for changes to requirements and design based on customer feedback

Following Iterative and Incremental practices means that developers start writing software earlier. Feedback from early prototypes helps to guide the direction of the project. Iterative and incremental development is covered in Chapter 2 [1, p. 17] of the textbook.

Agile

Computing has changed significantly since the first hardware became available. Consider that, within the span of one human lifetime, software development went from being an occupation that did not exist in the pre-computer era to one that currently employs millions of people. Not only are there more programmers, but there is, now, huge variety in terms of things like skill level and focus. It is only natural to expect that practices have evolved in response to such huge changes.

One of the most impactful advances in the evolution of software engineering practice has been the *agile* software development movement. The manifesto for agile software development[2] outlines 4 key values and 13 principles. Larman [1, p. 29] gives a brief overview agile methods.

The agile values are as follows,

- **Individuals and Interactions** over processes and tools
- **Working Software** over comprehensive documentation
- **Customer Collaboration** over contract negotiation
- **Responding to Change** over following a plan

[2] http://agilemanifesto.org/

The basic idea is that while everything on the list is important, the things on the left take precedence compared to the things on the right. For example, the fourth value gives us strong license to make adjustments to our plans as new circumstances arise.

There are many different agile processes. Some pre-date the manifesto while others came afterwards. Comparing agile processes with waterfall, the major differences are that agile projects are structured in such a way that the development activities can occur simultaneously and that the team benefits from more frequent feedback. The following sections cover specific examples of agile processes.

In addition to Chapter 2 [1, p. 17] of the textbook, which covers agile processes, we put together a list of readings: [44, 45]. Read these short articles in full.

3.2 Agile Processes

3.2.1 Scrum

Of the various agile software development processes, *Scrum* is the most popular. *The Scrum Guide* [46][3] is the definitive reference for Scrum. We only provide a brief outline here and expect the reader to follow up with a careful reading of the guide. Also, the article by Rising *et al* [32] gives a focused introduction to Scrum.

According to the creators, Scrum is a,

> Framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value (*The Scrum Guide* [46])

The name comes from the sport of Rugby where, at certain points during a game, teams form a large huddle. It is this image of cooperation and collective effort that the creators meant to evoke. Note that the quotation calls Scrum a "framework". This suggests that organizations need to adapt Scrum to their particular context. For example, there is nothing in Scrum that is specific to software development.

Scrum is defined by its *events*, *roles*, and *artifacts* and by the rules that govern them.

- Scrum Events
  - Daily Stand-up
  - Sprint
  - Sprint Retrospective
  - Sprint Review

[3] https://www.scrum.org/resources/scrum-guide

- Roles
  - Scrum Master
  - Product Owner
  - Development Team
- Artifacts
  - Product Backlog
  - Sprint Backlog
  - Increment

According to Brechner [47], all software development processes are essentially characterized by how they limit chaos. This is a useful notion to keep in mind when comparing the different agile processes. The key notion of Scrum is the *sprint*, and it is through the sprint that Scrum places limits on the inherent chaos of software projects. A sprint is a period of time during which all requirements and goals are frozen. This is also called a *timebox*. A typical sprint lasts between 2 and 3 weeks. By not allowing changes to the plan until the current sprint completes, Scrum prevents the development team from getting caught in a cycle of scope creep.

3.2.2 eXtreme Programming

As mentioned above, Scrum does not provide specific practices for software development. On the other hand, *eXtreme Programming* (XP), another agile process, has many useful practices. XP has been an influential agile process and is described at length in the book *Extreme Programming Explained* [8]. Speaking to the influence of XP, in a mid 2000s meta-analysis of agile software development research studies, Dyba *et al* [45] found XP was investigated "almost exclusively".

There are two key practices associated with XP: *pair programming* and *test-driven development* (TDD, also sometimes *test-first development*). In this section, we will focus on these practices. In our discussion of TDD, we will introduce the idea of *unit testing* and the popular JUnit tool for Java.

Pair Programming

Pair programming is the idea that all production code should be written by two people sitting at one machine. You can think of pair programming like taking a road trip: one person drives (they are behind the wheel) and one person navigates.

Andres and Beck [8] suggest the following benefits of pairing,

- Keep each other on task.
- Brainstorm refinements to the system.

- Clarify ideas.
- Take initiative when their partner is stuck, thus lowering frustration.
- Hold each other accountable to the team's practices.

(Andres and Beck [8])

Test-Driven Development

In general, it is very difficult (if not impossible) for a developer to know if a program is "correct" (i.e. does not contain any errors). To gain confidence in their software a developer can *test* it. Software can be tested at many different scales, from individual procedures to entire systems. Testing (typically) does not *prove* that software is without error, but it is a kind of meaningful evidence. With testing we want to see that the software behaves as expected under certain conditions. Another way to frame this is to ask what the correctness of the software implies. Therefore, we can think of testing as examining properties that are necessary but not sufficient.

A *unit test* is a check of the smallest piece of functionality in a software project: usually a single method or function. Other types of software tests are *integration tests* (run on a set of interacting objects) and *acceptance tests* (run on full systems). The main idea of TDD is that the unit tests are written *before* the components being tested. For example, following TDD, if a developer were writing a simple calculator application the first thing they would do is to code up some simple checks of the basic arithmetic operations. The most straightforward way to do this is to pepper the source code with a lot of print statements, see Listing 3.1.

Listing 3.1: Testing with print statements

```
Calculator calculator = new Calculator();

// Test the add method
System.out.println(calculator.add(12,8) == 20);

// Zero times a number equals zero
System.out.println(calculator.multiply(0,1) == 20);
```

We have all written code like this, especially if we are struggling to make things work in a limited amount of time. The impulse is correct, but there is much to gain by being a bit more systematic. One way to leverage features of Java to make testing easier would be to replace `println` with a method that takes a boolean and throws an exception if the value of the parameter is `false`. That way, verification of the test outcomes would not require manual parsing of the console output (we only need to check if any unchecked exceptions were thrown). We could go even further by adopting a unit testing framework like JUnit. A unit testing framework simply provides automation and reliable structure.

Note that the mere act of thinking about effective unit tests can already suggest non-trivial issues, even in this simple problem. For example, how should our methods behave in case of overflow? Or, is it important that our application maintain associativity of addition? Will this calculator deal with money? In that case, what is the strategy for rounding values?

By far, the most popular unit testing framework for Java is JUnit. This framework is integrated into Eclipse. Also, a standard maven Java project defines a source folder for JUnit tests. A typical JUnit test (still with the calculator example) is shown in Listing 3.2.

Listing 3.2: JUnit Test Example

```
import static org.junit.Assert.*;

import org.junit.Test;
import org.junit.*;

public class CalculatorTest {
   @Test
   public void testAdd() {
      Calculator calculator = new Calculator();
      double result = calculator.add(12,8);
      assertEquals(20,result,0);
   }
}
```

There are some important things to note in this example. First of all, `CalculatorTest` is a plain Java class. The *test* annotation tells us that it is a unit test. Secondly, the actual check is performed in the assertion, `assertEquals`. JUnit includes many different types of assertions. In this case, we want to see if `result` equals 20. The third argument is a way to specify an allowable error (for example, if we are dealing with floating-point calculations it may be enough to check our results up to some rounding error). Some other assertions are listed below, others can be found at the JUnit documentation[4].

- `assertArrayEquals("Message",A,B)`: checks the equality of the arrays
- `assertSame("Message",A,B)`: checks that the objects A and B are the same
- `assertTrue("Message",condition)`: asserts that condition is true
- `fail("Message")`: always cause a failure, can be used to indicate an incomplete test

Another, more complicated, example of a JUnit test class is shown in Listing 3.3. In this case, there are 2 unit tests that have a common setup. The setup

[4] `http://junit.sourceforge.net/javadoc/`

can be placed in a special method with a *before* annotation. The JUnit runner will automatically call `setUp` before each of the test methods. Moreover, the test methods are run independently. We need not worry about the state of `calculator` being carried over from one test method to the next.

Listing 3.3: JUnit Test With Setup

```
public class CalculatorTest {
  private Calculator calculator;

  @Before
  public void setUp() throws Exception {
    calculator = new Calculator();
  }

  @Test
  public void testAdd() {
    // Calculator calculator = new Calculator();
    double result = calculator.add(12, 8);
    assertEquals(20,result,0);
  }

  @Test
  public void testAdd1() {
    // Calculator calculator = new Calculator();
    double result = calculator.add(32, 1);
    assertEquals(33,result,0);
  }
}
```

As mentioned above, JUnit is integrated into the Eclipse IDE. Figure 3.1 shows what it looks when we run JUnit in Eclipse. If all of the unit tests pass then we see a green bar. If some of the unit tests fail the bar will be red.

3.2.3 Kanban

Like Lean (to be discussed in the next section 3.2.4), Kanban has origins in the Toyota Production System. Kanban, which means "signboard", in Japanese, is a method for visualizing workflow. Our primary source for Kanban is the book by Brechner [47]. There is a wealth of material available online on pertaining to Kanban. The popular project management tool *Trello*[5] offers an electronic version of a Kanban board.

In Kanban, features or work items are represented by small cards or sticky notes that are placed on a large board. The board can be a whiteboard or a corkboard. The board is divided into vertical columns. Each column represents

[5] `trello.com`

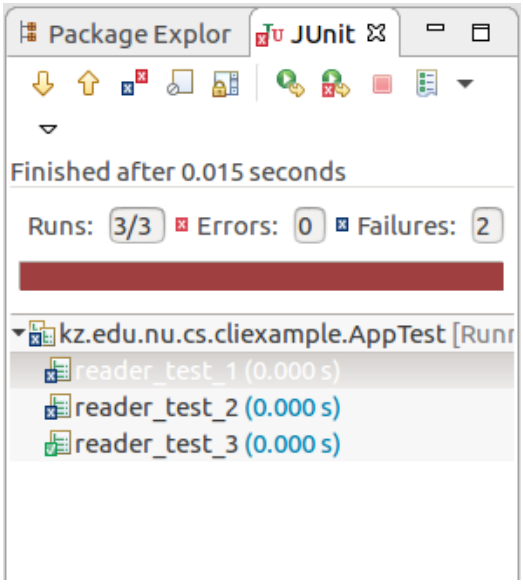

Figure 3.1: Running JUnit in Eclipse.

a stage of work. A simple scheme for naming the columns, in order of increasing completion and from left to right, is *backlog*, *specify*, *implement*, and *validate* [47]. The backlog plays the same as in Scrum. In specify, a feature is broken down into a set of smaller individual tasks. Implement is the actual work (coding, design, etc.). Validate is checked. This step could also include, for example, deploying the software artifact.

As work progresses, features flow across the board. Importantly, within each column there is a distinction between what is done and what is still in process. This distinction is represented by splitting a column in half. Also, there is a limit on the total number of cards that can be in a column at any one time. These are called WIP (work in progress) limits. For example, we are not allowed to advance a card if the WIP of the next column is already full. The WIP limits are how Kanban limits chaos. See the figure for an illustration of a Kanban board.

3.2.4 Lean

Required Readings — Poppendieck and Cusumano [33, Lean Software Development: A Tutorial].

To begin discussing Lean, we recall the story of the New United Motor Manufacturing Inc. plant (NUMMI) in Fremont, California[6]. In the 1980s, the world marveled at the success and efficiency of Japanese auto firms such as Toyota and Nissan. In terms of metrics such as labor hours to produce an automo-

[6]Shook `https://sloanreview.mit.edu/article/how-to-change-a-culture-lessons-from-nummi/`

bile, the Japanese were performing significantly better than the Big Three US automakers.

American firms sought to emulate the successful methods of their Japanese counterparts. One expression of this was the NUMMI assembly plant. NUMMI was a joint venture of Toyota and General Motors (GM). The idea was to apply the Japanese practices on US soil. Also, for NUMMI, the joint venture was implemented at an existing GM plant. The terms of the joint venture said that NUMMI had to retain the previous workforce even though it did not have the best reputation within the GM organization. For example, there were high rates of absenteeism and a recalcitrant union culture. Management found that the application of Lean techniques, which strongly emphasized employee involvement, effected a huge positive transformation. The NUMMI plant, previously one of GM's poorest performing, became one of its best. More recently, the Toyota-GM venture ended (in 2010) and the plant is now owned by Tesla.

Subsequently, practitioners in domains outside of manufacturing began to recognize the value of Lean thinking, and Lean was applied to software development. The Poppendiecks [39] were trailblazers in this regard. Broadly speaking, Lean development focuses on the value delivered to the customer and is codified in seven principles that we restate below.

- optimize the whole
- eliminate waste
- build quality
- deliver fast
- engage everyone
- keep getting better

The tutorial [33] includes a short elaboration on each of the seven principles.

3.3 Requirements Engineering

Required Readings — Larman [1, Ch. 5, Evolutionary Requirements].

Properly understanding and documenting requirements is a crucial part of a software development project. The following quote is a brief definition.

> Requirements form a set of statements that describe the user's needs and desires. (Tsui *et al* [6])

The process of collecting requirements from stakeholders is called *requirements elicitation*. We can further distinguish between *functional* requirements and *non-functional* requirements: *functional* requirements say what the system will do while *non-functional* requirements describe what the system won't do. In this course, we will employ *use cases* as the primary method of document requirements.

3.3.1 Use Cases

Required Readings — Larman [1, Ch. 6, Use Cases].

> we organize our experience and our memory of human happenings mainly in the form of narrative—stories, excuses, myths, reasons for doing and not doing, etc.
>
> (Bruner [48], cited by Van Every and Taylor [49])

> every real story 'contains, openly or covertly, something useful'
>
> (Benjamin [50], quoted by Greenblatt [51])

Stories are fundamental to how we make sense of the social world. It is not surprising that stories have also become an important component of requirements engineering. As software developers, the stories that we care about are descriptions of how the user interacts with our systems. These stories are called *use cases* or *user stories*.

A use case is a description of a sequence of interactions between a system and an external actor. The actor derives some benefit or achieves some goal by interacting with the system. An *actor* can be a user or another external system (e.g. credit card or payment processor, external database). A *scenario* is a specific path through a use case. There is usually one intended path through a use case, this is called the main success scenario. Additionally, there can be paths that include alternative flows and exceptions.

Use cases are covered at length in the textbook [1, Ch. 6]. It is important to reiterate a significant point. Use cases are, primarily, textual. That is, use cases are about writing. While UML includes a class of Use Case diagrams, recording use cases should not be an exercise in diagramming.

A template for use cases is shown in Listing 3.4. The textbook elaborates on the meanings of the sections.

Listing 3.4: Use Case Template

```
<name of the use case>

Context: <statement of the goal>
Scope: <the system under consideration>
Primary actor: <description or role>
Stakeholders and Interests: <stakeholders and relevant interests>
Precondition: <expected starting conditions>
Min. Guarantees: <protection of the interests>
Success Guarantees: <state of world if goal succeeds>
Trigger: <initiating event>

Main Success Scenario: <steps of main scenario>
  <step #> <description>

Extensions: <extensions ref. main scenario>
  <step altered> <condition> <action description>

Related Information:
  <other important information for the project>
```

Chapter 4

Developing for the Web

Most people interact with websites on a daily basis. The web has blended seamlessly into the environment, and we are no longer so mindful about its presence. Also, the web is becoming a primary platform for many application experiences that were previously the domain of desktop applications[1]. Many software development positions center on the web. At a basic level, much of what concerns developers in industry is using the web to interface between the user and some persistent data storage (like a mysql database).

The purpose of this chapter is to provide an overview of the technologies that we are using in this course. These are the technologies that you will need for the course project. As stated in the introductory chapter, we expect basic proficiency with the Java [2]. We will review other tools that you may not have experience with such as JUnit and maven. The web application material appears later in this chapter.

One should note that there are a huge number of frameworks, servers, and technologies for web development. Even with more research-oriented languages/concepts you can quickly locate an associated web framework[2]. Similarly, in the front-end and Javascript spaces the number of frameworks and tools is truly mind boggling.

4.1 Builds and Maven

Maven is a tool for building and managing Java-based projects. By a *build* we mean an operational version of a software product. As long as the development workstation is on the network, maven provides easy access to a large repository of useful Java libraries. Maven greatly simplifies the process of including dependencies compared to manual manipulation of the classpath or `jar` files.

[1] One example of this is how services like gmail have replaced dedicated email clients for many users. Another example is Intuit, who moved their popular tax preparation software product *TurboTax* to the web.

[2] For example, see the Yesod web framework for Haskell (`www.yesodweb.com`) and the Ur/Web language based on dependent types (`www.impredicative.com/ur/`)

The configuration of a maven project is stored in a special XML file named `pom.xml` (for *project object model*). The presence of `pom.xml` in the root directory of a project indicates that it is, in fact, a maven project. In addition, the layout of directories in a maven project is organized in a standard fashion[3]. The standard maven layout[4] is shown in Listing 4.1. Note that some of these folders are not required. For example. if we are not writing a web application then we probably will not have a directory `/src/main/webapp/`.

Listing 4.1: Maven Project

```
/Project Root/
  |--/src/
  |     |--/main/
  |     |     |--/java/
  |     |     |--/webapp/
  |     |--/test/java/
  |--README.txt
  |--LICENSE.txt
  |--NOTICE.txt
  |--pom.xml
```

Maven can be operated either on the command line or through an IDE plugin (Eclipse supports maven). The maven command line tool is named `mvn`. Maven is organized around build lifecycles: well-defined processes for building the project. For example, the `default` lifecycle comprises a process for going from source code to a deployment artifact (for example, a `jar` file). A lifecycle is made up *phases*, and phases are made up of *goals*. An example of a goal is `compile` (compiles the source code). To run the compile goal against the project you would navigate to the root directory and type `mvn compile`. Please review the maven documentation covering the basics[5]. Maven is an Apache project and is open source.

Maven simplifies the process of pulling in *dependencies*. For the developer, that is a primary benefit. In this context, a *dependency* is some other Java code that you are using in your project. For example, Apache Commons[6] is a very popular project consisting of open source, reusable Java components. Using maven, a developer can quickly include third-party libraries such as those in Apache Commons. If required, the maven tool will automatically download and cache the required libraries from the public repository. In maven, libraries are identified by a `groupId`, an `artifactId`, and a version number. To include the dependency the developer modifies (adds to) `pom.xml`. In Listing 4.2 we show an example. In this case, we are including `commons-cli` (utilities for writing

[3]The philosophy of maven is "convention over configuration". What this means is that, while a project can be customized, maven provides sensible defaults.

[4]https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

[5]https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

[6]https://commons.apache.org/

command line applications) and `commons-text` (utilities for string processing). One of the exercises asks you to create a maven project and use these specific libraries.

Listing 4.2: Example: Dependencies in project object model

```
<dependencies>
  <dependency>
    <groupId>commons-cli</groupId>
    <artifactId>commons-cli</artifactId>
    <version>1.4</version>
  </dependency>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-text</artifactId>
    <version>1.2</version>
  </dependency>
</dependencies>
```

4.2 git

Required Readings — Github [52, Github Git Cheat Sheet]. Familiarize yourself with all commands shown in the cheat sheet. You may need to consult man pages as well as the git documentation.

While developing the Linux kernel, Linus Torvalds created `git`, a version control system. This tool has since become widely adopted in the software engineering community. With git, you can either work locally or collaboratively, and maintain a history of all edits in a project. The command line tool is named, simply, `git` and is available online[7].

If we look at the full range of features, git is quite complex. Git can feel overwhelming because of the large number of features and terminology. Even contributors to the git project itself can find the system and documentation confusing. If you are getting started with git it is best to become familiar with the most common commands and practice. In this section, we will introduce basic concepts and common commands.

Git projects are usually called *repos* (for *repository*). You can create a new git project in one of two ways: call `git init` in the root directory of your project or clone an existing repository with `git clone <url>`. A project in Git is managed by a special directory called `.git` in the root of your project.

Git uses a two-stage commit process, changes are moved first to a "staging area", and are, later, committed with the `git commit` command. The `git status` command will show the current state of your project: changes that have been staged but not committed, files that have been modified but not staged.

When you are working on your project, you make edits as you would normally. When you reach a point where you want to mark your progress use `git`

[7] `https://git-scm.com/`

`add <file>` to stage the changes you made. The staged changes can be committed by running `git commit -m "<message>"`. At any point in the future, the project can always be rolled back to previous commits. The command `git diff` tells you the changes that have been staged but not committed. Note that git does not automatically track every file in your project. For example, if you create a new file in your project then `git status` will warn about untracked files. To track the new file, run `git add <new file>`. It is probably best practice to track all the files in a project but there are some files that you should not track. Examples are binaries, temporary files created by the OS or and IDE, and configuration files that contain sensitive information. A `.gitignore` file is a list file paths that you explicitly do not want to track.

In git, parallel work can proceed in separate branches. To see the current branch you can run `git branch`. The command `git branch <branch name>` creates a new branch. The `git checkout <branch name>` command switches from the current branch to `<branch name>`.

Git enables collaboration through *remote* repositories. On a team, each developer keeps their own copy of the project locally. These distributed copies are synced by means of a central copy called a remote. Developers can interact with the remote through certain git commands. Changes made by others can be retrieved from the remote by issuing a `git pull` command. You share your update by pushing them upstream using `git push`.

Finally, Github[8] is a web application for project management built on top of git. Github provides free and paid hosting for git repositories. In addition to hosting Github also offers project management and social features. The main collaboration tool offered by Github is the *pull request*. A pull request is a request to the project owner to accept changes made by a collaborator. Review the Github documentation on pull requests[9]. In addition to pull requests, Github also offers support for *issues* and a social network.

4.3 Basic Command Line and Shell Scripting

A shell is an environment for running commands, programs and shell scripts and inspecting the system and processes. This is the Unix command line that you should have encountered many times already. Be aware that there are different versions of the Unix shell: Bourne shell, Bash shell, Korn shell, C shell, etc. Here, we cover the Bash shell which is standard on Linux distributions and Mac OS.

A *shell script* is a list of commands given in order of execution. All of the basic features that we expect from other languages are available, however their usage or syntax may appear strange at first. Some of these features would include: comments, conditionals, control flow, functions, and variables.

Some of the tasks we might perform with scripting include,

[8] `github.com`

[9] `https://help.github.com/articles/creating-a-pull-request/`

- Automate copying of files, FTP, ...
- Searching or changing the filesystem
- Automating or composing programs that use the standard input and output streams, `STDIN`, `STDERR`, and `STDOUT`

To bolster your knowledge of the command line you should review the material available from *The Linux Documentation Project*, especially the *Command Line Tools Summary*[10]. At minimum, you should understand input/output streams, pipes and filters, and basic text processing at the command line.

4.4 Regular Expressions

Regular Expressions, also called *regex*, are a kind of syntax for searching for patterns in strings. Most languages, including Java and Javascript, have some support for regular expressions and the notation is generally consistent. A regular expression defines a (potentially infinite) set of strings.

In Java, we use the `java.util.regex.Pattern` and `java.util.regex.Matcher` classes to work with regular expressions. Listing 4.3 shows a some simple regular expression code in Java. The regular expression is the string `[abc]+` passed to the static method `compile()`.

Listing 4.3: Example: Regular Expression Syntax and Classes in Java

```
Pattern pattern = Pattern.compile("[abc]+");
Matcher m = pattern.matcher("aabba");

m.matches(); // returns true for match
```

A string is the simplest regular expression, and represents a search for a fixed pattern. For example, if we want to find the occurrences of the word 'hello' in a sentence then we define a regex `"hello"`. The matcher object has two methods that you should remember.

- `.find()`: search the input for a pattern
- `.matches()`: attempt to match the entire input against the pattern

Listing 4.4 shows how you could search a string for a fixed pattern.

Listing 4.4: Example: Fixed Pattern Regular Expression

```
pattern = Pattern.compile("hello");
Matcher m = pattern.matcher("Can you say hello?");

// prints From 12 to 17
if (m.find()) {
```

[10] `https://www.tldp.org/LDP/GNU-Linux-Tools-Summary/html/index.html`

```
    System.out.println("From " +
                       m.start() +
                          " to " +
                          m.end());
}
```

There is more to regular expressions than finding a fixed pattern in a string. The power of regular expressions is that we they provide a concise way to refer to a large class of strings. Some of the most important features are given in the following list.

- *Alternatives*: one thing *or* something else, indicated with vertical bar `|` or square brackets
- *Character classes*: express common ranges of characters (lowercase letters, digits), typically indicated with square brackets
- *Quantification*: express how many times a pattern should occur
- *Grouping*: extract sub-patterns in a match

Some common predefined character classes are explained in the following list. These will match to a single input character. Character classes can be negated with the caret symbol `^`.

- `[0-9]`: digits
- `[a-z]`: lowercase letters
- `[A-Z]`: uppercase letters
- `[a-zA-Z]`: upper and lower case letters
- `.`: matches any single character

The next list shows how we quantify patterns, that is, indicate how many times they can occur. In regex, `?`, `*`, and `+` are metacharacters. 'A' just means some other pattern specified as a regex.

- `A*`: Kleene star, 'A' occurs zero or more times
- `A+`: 'A' occurs one or more times
- `A?`: 'A' occurs zero or one times
- `A{n}`: 'A' occurs n times

Grouping is also an essential part of applying regular expressions. For example, we might be processing names and want to extract first and last names every time we find a match for a full name. Or, if we are working with phone numbers than we might want to quickly extract the area code or country code. *Capture groups* allow us to capture part of a match. Capture groups are indicated with parentheses and the matching substring can be retrieved with the `group(int a)` method on the matcher object.

Listing 4.5 shows examples of using regular expressions in Java that illustrate the ideas above.

Listing 4.5: More Java Regex

```
Pattern pattern;
Matcher m;

// matches bit, bat, and bot
pattern = Pattern.compile("b[ioa]t");

// same as above
pattern = Pattern.compile("b(i|o|a)t");

// match any single character except a,b, or c
pattern = Pattern.compile("[^abc]");

// pattern for posix punctuation class
pattern = Pattern.compile("\\p{Punct}");

// match any character any number of times
pattern = Pattern.compile(".*");

// match one or more digits followed by three letters
// e.g. 01234tuv
pattern = Pattern.compile("[0-9]+[a-zA-Z]{3}");

// two, possiblyt empty sequences of digits separated
// by 0 or 1 lowercase letters
// 11111a2222, a22222, 11111a all match
pattern = Pattern.compile("[0-9]*[a-z]?[0-9]*");

// parsing path strings with regex
pattern = Pattern.compile("/([a-z]+)/([0-9]+)");

m = pattern.matcher("/abc/001");
m.matches();

// prints abc
System.out.println(m.group(1));
```

```
// prints 001
System.out.println(m.group(2));
```

In summary, regular expressions are a powerful notation for expressing patterns. However, it is important to remember not to overdo them. The concise notation of regex comes at the expense of being difficult to parse. You should not expect your fellow developers to be able to instantly grasp the meaning of your regex laden code. Also, pay special attention to potential corner cases, it is easy to write a regular expression that you think is doing one thing but that is actually doing something else.

4.4.1 Exercises

1. Suppose you want to match dates written in a standard format (for example, `10/11/12` for November tenth 2012), describe at least three potential problems with the following regular expression as a solution: `[0-9]{2}/[0-9]{2}/[0-9]{2}`.

4.5 Java Enterprise Edition

Required Readings — Highlighted sections of the Java EE Tutorial [53]. Tomcat User Guide [54].

For the team project, in this course, you will develop a web application based upon Java technologies. The Java server-side technology is called Java EE (Enterprise Edition). It is important to note that Java EE is a specification and a developer/team must choose a specific implementation. Some implementations of the EE specification may be more complete than others. The following section is based on Apache Tomcat, which implements part of Java EE, namely the Servlet API.

Of course, Java is not the only choice of programming languages for server-side software. CGI scripting (Common Gateway Interface) was an early web standard that worked with languages such as Perl. Popular web servers like Apache have also offered native-language APIs for C and C++. More recently, with *node.js* [16], server-side Javascript has become fashionable. Historically, the Java Servlet API became popular because there was already a large community of Java developers and the technology made code more easily portable.

Java EE is an extension of Java Standard Edition for server-side applications. Apache Tomcat[11] is a production grade, open source web server that implements part of the EE specification. Tomcat is a *servlet container*. A *Servlet* is a Java program that can respond to HTTP requests. Servlets are defined in `javax.servlet` package.

We have two primary references where you will read about Java EE technology: the Java EE Tutorial [53] and the Tomcat User Guide [54]. Note that the examples in the Tutorial [53] are based on NetBeans and GlassFish, which is the reference implementation of the EE specification. We have elected to

[11] Available at `tomcat.apache.org`

use Tomcat for this course since it is more lightweight. The readings from the tutorial are meant to familiarize with Java EE in general. Also, the tutorial stretches to nearly 1000 pages. So, do not feel the need to read it cover-to-cover, as it were, focus on the assigned sections.

You should carefully review the following sections of the Java EE Tutorial [53].

- **Chapter 1: Overview**: High-level overview of Java EE technologies
- **Chapter 6, Sections: 6.1, 6.2, 6.4, 6.5**: Overview of web technologies
- **Chapter 13, Sections: 13.1, 13.2**: Ajax
- **Chapter 17**: Servlets
- **Chapter 19**: JSON
- **Chapters 27 and 29**: RESTful web services

4.6 Javascript

Javascript has seen significant growth over the past decade. Javascript initially became important mainly for the reason that it is the language of the web browser. JS was designed by Brendan Eich in approximately 10 days and was added directly to Netscape Navigator (an early web browser and the precursor of firefox). His goal was to provide a language for adding interactivity to simple web pages. The history of the Javascript language is quite interesting, and it is recounted in *The Good Parts* by Douglas Crockford [55] (the primary advocate of the JSON format and inventor of the static analysis tool `jslint`).

As stated above, Node is a JS runtime. This allows us to write JS outside the context of the browser. A basic "Hello, World!" program for node is shown below. There are some things to note here. First is that `console.log` is the common way to write to the console. The first line of the program is called a *hashbang* or *shebang*. It is not part of Javascript. This line tells the operating system which interpreter to use to execute the script. It should be the path to node on your system.

Also note that we have an *immediately invoked function*. This script defines a function `say`. The parentheses wrapping the function definition and the parentheses before the semicolon cause the defined function to be invoked (called) immediately. This is a common pattern in JS and a potential confusion.

Listing 4.6: JS

```
#!/usr/local/bin/node

(function say() {
  console.log("Hello world!");
})();
```

Node can be run in two different ways: either from the command line or interactively with the REPL. Pass a filename as an argument to have node execute it: `node yourfile.js`. The command `node` by itself will put you in the REPL (quit the REPL with `.exit`)

4.6.1 Javascript Language

This section contains a brisk review of the Javascript language. There are many peculiarities to the language that the developer should be aware of. One example that is often cited is `0.1 + 0.2`. Typing this expression into the node REPL one obtains `0.30000000000000004` instead of the expected 0.3. This is because JS uses the IEEE 754 floating point standard which cannot perfectly represent simple fractions. In fact, all numbers in JS are floating point. There is no separate integer data type in the language.

4.6.2 Functions

Functions are declared with the keyword `function`. Variable and function declarations are hoisted, meaning they are automatically moved to the beginning of their blocks. For this reason the function `say` can be called before it is declared in the example below. Only variable declarations are hoisted, an assignment will happen where it appears in the code.

Listing 4.7: JS

```
y = "hello : y";

say();

function say() {
  console.log(y);
  console.log(x);

  var x = "hello : x";
  // var y;
};
```

A *closure* or *lexical scoping* simply means that a function has access to the parent scope. The important point to remember is that the scope of a function can still exist even after it has returned, as in the example below.

Listing 4.8: JS

```
var prvExample = (function () {
    var somedata = "Private Data";
    return function () {
      somedata = somedata + "!";
      return somedata;}
```

```
})();

console.log(prvExample());
console.log(prvExample());
```

Javascript is a fully functional programming language. Functions are *first class*. This means that we can pass a function wherever we can pass an object, value, or array. The essential parts of a function are its arguments and its return value. A function need not be named. Functions without an identifier are called *anonymous*.

An expression with a free variable like $x^2 + 2x + 1$ can become a function in a process called *abstraction*. *Application* is the process of calling the function, usually this means to substitute some value or expression for a free variable. In the *lambda calculus* abstraction is indicated by $\lambda x.t$ and application is written without any parentheses or commas. To apply the expression f to a we would write $f\ a$. For $\lambda x.t$ we say that x is *bound* in t. The lambda expression $\lambda x.x^2 + 2x + 1$ has the same meaning as the function in the example below.

Listing 4.9: JS

```
var pl = function (x) {
   return x * x + 2 * x + 1;
}
```

The javascript runtime does not check parameters for function calls. This means that, once a function is declared, it can be correctly called with any number of arguments or none at all. The onus is on the programmer to guard against errors and to pass the proper arguments. There is a predefined object called `arguments` in the body of any JS function that is demonstrated in the example below.

Listing 4.10: JS

```
function sayArguments() {
  console.log(arguments);
}
```

Similar to `arguments`, function invocations also get an implicit parameter called `this`. This is the *context* in which the function was called. Its behavior is more complicated than in object-oriented languages where `this` usually refers to the current object. In JS, the value of `this` depends on how the function was invoked. There are 4 ways: function, method, constructor, `apply` and `call`. Some of this is demonstrated in the following example.

Listing 4.11: JS

```
function say() { console.log(this);}
var o1 = { name : "value" )};
var o2 = { fn : say };
```

```
say();          // prints global object
o2.fn();        // prints o2
say.call(o1);   // prints o1
```

A function becomes a *constructor* when it is invoked using the keyword `new`. This syntax is intended to achieve an object-oriented coding style and be more familiar to Java programmers. The code example below shows how we might define a class for a 2d point in JS. When invoked as a constructor `this` will refer to the object being constructed.

Listing 4.12: JS

```
function Point() {
  this.x = 0;
  this.y = 0;

  this.translate = function(tx, ty) {
    this.x = this.x + tx;
    this.y = this.y + ty;
  }
}

var p = new Point();
```

Scoping in Javascript works a bit differently than in other common languages. A variable can be declared with a `var` or `let` statement. Without a declaration keyword a variable is assumed to be global. The keyword `let` is a recent addition to the language to allow block scope.

Listing 4.13: JS

```
var i = 12345;

for (var i = 0; i < 10; i++) {}

console.log(i); // '10'

for (let i = 0; i < 100; i++) {}

console.log(i); // '10'
```

More explicit lambdas have been added to Javascript. This uses an arrow syntax and, arguably, allows for the definition of anonymous functions in a cleaner style.

Listing 4.14: JS

```
var x = (a, b) => a + b;
```

```
var x = (a, b) => {return a + b;};

// basically equivalent to
var x = function (a, b) { return a + b;};
```

A *callback* is a function to be executed at a later time. Some examples of where callbacks are encountered are the asynchronous filesystem operations in the core node module `fs`. Another example that we can discuss for illustration is array sorting. Any array has a `.sort` method that will return a sorted array. A custom comparison operation can also be given as a callback as in the next example.

Given an array of strings, the default behavior of the `.sort()` method will sort in dictionary order. However, we may want to sort the array in another way. For example, sort only the valid integer strings and ignore cases where the string is not an integer.

Listing 4.15: JS

```
var arr = ['10', '-1', 'mmm234', '1a', '1b', '1c', '2', '1000'];

console.log(arr.sort());
// [ '-1', '10', '1000', '1a', '1b', '1c', '2', 'mmm234' ]

// arr.sort(customSortB) sorts as
// [ '-1', '2', '10', '1000', '1a', '1b', '1c', 'mmm234' ]
function customSortB(v1, v2) {
  var pattern = /^(-?)[0-9]+$/;

  if (pattern.test(v1) && pattern.test(v2)) {
    return parseInt(v1, 10) - parseInt(v2, 10);
  } else if (!pattern.test(v1) && pattern.test(v2)) {
    return 1;
  } else {
    return -1;
  }
}
```

Functions in JS are special kinds of objects which implies that they can have methods and properties attached. The `.call` and `.bind` methods from earlier are examples but we can attach our own methods and data to functions. *Memoization* uses this to cache the values of an expensive computation, so that the function need only compute it once.

The function `frozen` shows an example of this pattern. The function will return a random number the first time it is called for a given argument value. All subsequent calls with that argument value will return the same value.

Listing 4.16: JS

```
function frozen(n) {
  if (!frozen.ans) {
     frozen.ans = {};
  }

  if (frozen.ans[n] !== undefined) {
```

```
    return frozen.ans[n];
  }
  c = Math.floor(10 * Math.random());

  frozen.ans[n] = c;
  return c;
}
```

4.6.3 Recursion

Recursion is the name given when a function or procedure calls itself. A functional programming style favors recursion as an alternative to looping and some kinds of program control flow. Sometimes, recursion can lead to a cleaner or easier to understand solution but recursive programs can also be difficult to reason about. Depending on how the language itself is implemented a recursive solution can also have performance drawbacks compared to an imperative solution (this is the topic of *tail call optimization*).

A *tail call* is a special kind of recursive function or function call. A function call is in tail position if it is the last thing that happens before returning. For recursion in tail position compilers can make an optimization that turns the recursion into iteration: *tail call optimization* (TCO). The point is that, with TCO, the runtime does not need a new stack frame for each recursive call.

TCO is a recent addition to Javascript, introduced in ECMAScript 2015 (ES 6). In node we can see this directly by running in strict mode with the harmony flag (`node --use_strict --harmony`). This feature allows for very deep recursion and certain functional programming parameters that would not otherwise be possible.

The code listing below show two recursive functions that perform an equivalent computation. The recursive call in `recursion1` is in tail position and can be optimized. Imagine what happens when either of the functions are called with a large value of `n`.

The function `rec1_tco()` shows (approximately) how an optimizing compiler will interpret `recursion1`. The parameter values are changed and the function is restarted. With the flags mentioned above `recursion1` and `recursion2` will behave differently (try this).

Listing 4.17: JS

```
// in tail position
function recursion1(n) {
   if (n === 0) {
      return 0 + 10;
   }

   return recursion1(n - 1);
}
```

```
// not in tail position
function recursion2(n) {
   if (n === 0) {
      return 10;
   }

   return 0 + recursion2(n - 1);
}

// approximate function
// with tail call optimization
function rec1_tco(n) {
   while(true) {
      if (n === 0) {
         return 0 + 10;
      }

      n = n - 1;
   }
}
```

Euclid's Algorithm for computing the greatest common divisor of two numbers is a classic example of a recursive solution to a problem. A Javascript implementation is shown in the next example.

Listing 4.18: JS

```
function gcd(x, y) {
  if (y === 0) {
    return x;
  }
  return gcd(y, x % y);
}
```

In their algorithms textbook, Sedgewick and Flajolet [56] draw a parallel between recursion and proof by induction. To prove a statement (for instance, about the natural numbers) by induction requires two parts: a base case and an induction step. First we prove that a property holds in the base case, and then we prove that the property is preserved when taking successor. Then the statement or property is proved for all cases by induction.

Likewise, when writing a recursive function we need a way to ensure that the computation eventually terminates (base case) and we also need to to divide a big problem into smaller problems (recursive call). Both of these properties are apparent in the greatest common divisor implementation.

4.6.4 Strings

Javascript strings are sequences of Unicode characters. There is no separate `char` datatype in JS. Each string has a `length` property. String literals can be indicated with either single or double quotation marks (escape with \' or \").

Listing 4.19: JS

```
// string with unicode character
var s = "\u03BB abs";
s.length; // 5;

console.log(s);
```

There are a number of useful methods available for string objects.

- `.substr(i, j)`: Extracts substring starting at `i` of length `j`
- `.slice(i, j)`: Extracts substring statring at `i` and ending at `j`
- `.split(",")`: Return an array from string with separator character

Listing 4.20: JS

```
var s = '1,22,333,4444,55555';

s.split(",");
// returns
// [ '1', '22', '333', '4444', '55555' ];
```

4.6.5 Arrays

Arrays are a common data type in programming. In JS arrays are just special objects and thus have access to methods. This chapter will give a brief overview of some of the programming constructs that are available for manipulating arrays in JS. A thorough and cogent treatment of JS collections and arrays can be found in Chapter 9 of [57]. In this section we talk about: creating and manipulating arrays, iteration, map and reduce operations, and applying a predicate.

Arrays can be created with either the literal notation `[]` or with the built-in `Array` constructor and have a property `length` that specifies the size of the array. The first element is at index 0 and the last element is at `arr.length - 1`. There are no out of bounds warnings if we try to access elements beyond the length, the system simply returns `undefined`. Assigning to a position beyond the bounds will appropriately expand the array. The length will be updated and intermediate values will be filled in with `undefined`.

The next example demonstrates four simple methods for adding and removing elements from arrays.

Listing 4.21: JS

```
var people = [];

// Pushes to end of array
people.push("Joe");
people.push("Jim");

// Adds new item to beginning of array
people.unshift("Mary");

// Removes item from end of array
// and returns it
var x = people.pop();

// Removes first item and shifts
// remaining elements
var y = people.shift();
```

The keyword `delete` will remove an element at an index but it will leave behind undefined. The `splice` methods allows us to remove elements in a range and takes two arguments: the start position and the number of items to remove.

Listing 4.22: JS

```
var arr1 = [1,2,3,4,5];
delete arr1[2];

console.log(arr1); // [ 1, 2, , 4, 5 ]

var arr2 = [1,2,3,4,5];

arr2.splice(1,2) // returns [2, 3]
console.log(arr2) // [1, 4, 5]
```

Javascript now provides a method `forEach` on arrays that will iterate through the elements of an array and apply a callback each time. The next example shows iteration with a for loop and with `forEach`.

Listing 4.23: JS

```
var arr = ["Mary", "Bob", "Alice"];

// typical iteration
for (let i = 0; i < arr.length; i++) {
   console.log(arr[i]);
}
```

```
// forEach approach
arr.forEach(function (person) {
   console.log(person + "!");
});
```

Map and *Reduce* are familiar operations from functional programming languages or libraries. For example, these operations were introduced to Java with the *streams* library. *Map* means to apply a callback over each element of an array and return all of the results as a single new array. *Reduce* means to repeatedly apply an operation to the elements of an array that results in a single value. Examples are shown in the next code listing. With `reduce` we need to supply a callback and an initial value.

Listing 4.24: JS

```
var arr = ["Mary", "Bob", "Alice"];

arr.forEach(function (person) {
   console.log(person + "!");
});

var mappedArr = arr.map(function (x) { return x[0];});
// [ 'M', 'B', 'A' ]

var n = [1,2,3,4,5,6,7];

console.log(n.reduce(function(acc, nn) {
   return acc + nn;
}, 0)
);
```

4.6.6 Synchronous vs. Asynchronous

Node.js uses an *asynchronous* model of communication. The speed of node for I/O applications is often attributed to this. In an I/O based application, processes will spend most of their time waiting for an operation to return: retrieving data from the disk, requesting resources from the network, connect to a database etc. Multithreaded approaches incur some overhead for each process or thread they need to launch.

The following example of asynchronous and synchronous requests comes from [55]. The client will halt until the `reqSync()` function returns. We can also say that `reqSync()` is a blocking call. The same kind of request is also shown in an asynchronous style. Note that the function `reqAsync()` will return immediately. When the request is completed the result will be processed by the anonymous function that was passed.

Listing 4.25: JS

```
// synchrononous style
request = prepareReq();
response = reqSync(request);
display(response);

// asynchronous style
request = prepareReq();
reqAsync(request, function (response) {
    display(response);
    });
```

4.6.7 Importing Code

The example below demonstrates key features of node such as the global object and the syntax for importing modules. The statement `var fs = require("fs");` is the equivalent of an import statement in Java (CommonJS). Note that some modules such as `fs` (the filesystem module) are built into node. These are core modules. Other modules, such as the `chalk` module for ANSI coloring, need to be installed before you can import them.

Listing 4.26: JS

```
var fs = require("fs"); // core module
var chalk = require("chalk");
var user = global.process.env.USER;

console.log(user);
console.log(chalk.green('~~~'))

if (fs.existsSync("erlngb.py")) {
  console.log("file exists");
} else {
  console.log("file does not exist");
}
```

4.7 Front-end Development

Front-end development refers to the user-facing, client-side component of a web application. The front-end is written in HTML, CSS, and Javascript. As mentioned above, the browser is fairly ubiquitous. In the past, the differences between the various browsers (IE, Netscape) was more of a challenge than it is today.

The architecture of the web is known as REST, which stands for *Representational State Transfer* and is described in a very famous PhD thesis by Field-

ing [58]. REST has many important characteristics: client-server, stateless, resources accessed by making HTTP requests, and URLs (not an exhaustive list). REST is based on the idea that the client always sends everything that the server needs to service the request. This property was critical as the web scaled. From the point of view of a web application developer, it is important to understand REST because of the popularity of RESTful APIs. Most online services such as social media and search providers expose their content through a RESTful API to enable machine-to-machine communication. RESTful APIs also enable dynamic applications in the browser, and many front-end frameworks such as Angular and Backbone are implicitly structured around the idea of a RESTful backend.

In the early days of the web, most of the content was static HTML pages that were linked together with anchor tags. Users interacted with the server by making sequences of HTTP requests. And if an application needed to process data from a user a form was returned that the user could fill in. Submission of the form would issue a POST request to the server. While this is a powerful paradigm for distributed applications, people wanted more interactivity in their web pages, something closer to the desktop application experience.

The first step to enable dynamic web applications was the introduction of scripting in the browser. This was first done on the initiative of browser vendors. Eventually, scripting in the browser was standardized by the European Computer Manufacturers Association (this is why people sometimes say that the "official" name of Javascript is ECMAScript).

It is important to recognize that Javascript is just a programming language. To manipulate web content on the client scripts call into the special DOM API. DOM stands for *Document Object Model*. The DOM is a tree-like structure that describes the contents of a web page. The API gives the front-end developer access to what is actually displayed by the browser. The most popular Javascript library, jQuery, is basically a more programmer friendly wrapper around the native DOM API of the browser.

The second important innovation for realizing a desktop application experience in the browser was Ajax, Asynchronous Javascript and XML. Ajax is a technology that allows the browser to make additional requests to the server after a page loads. For example, news websites often allow readers a space to comment on the articles. If a user navigates to a particular article there may be a link at the bottom of the page that says something like "Load Comments". Clicking the link will open up, say, the 10 most recent comments dynamically (meaning the user sees no page refresh or navigation). Sometimes news websites even notify in real-time as new comments are posted on an article. Another common pattern that is made possible by Ajax is real-time suggestions as a user types, in a search box for example.

The above examples are made possible by Ajax. In essence, the client side script is able to request additional information from the server "behind-the-scenes". To make an Ajax call, a web developer interacts with the built-in `XMLHttpRequest` object.

Bibliography

[1] C. Larman, *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development*. Prentice Hall PTR, 2012.

[2] "Java tutorials: Oca i preparation." [Online]. Available: https://docs.oracle.com/javase/tutorial/extra/certification/javase-8-programmer1.html

[3] K. El Emam and A. G. Koru, "A replicated survey of it software project failures," *IEEE software*, vol. 25, no. 5, 2008.

[4] M. Laitinen, M. Fayad, and R. Ward, "Software engineering in the small," *IEEE Software*, vol. 17, no. 5, p. 75, 2000.

[5] E. W. Dijkstra, "The humble programmer," *Communications of the ACM*, vol. 15, no. 10, pp. 859–866, 1972.

[6] F. Tsui, O. Karam, and B. Bernal, *Essentials of software engineering*. Jones & Bartlett Learning, 2016.

[7] P. McBreen, *Software craftsmanship: The new imperative*. Addison-Wesley Professional, 2002.

[8] K. Beck, *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[9] P. F. Drucker, *The New Realities*. Harper and Row, 1989.

[10] R. A. Lanham, *The electronic word: Democracy, technology, and the arts*. University of Chicago Press, 2010.

[11] d. boyd and K. Crawford, "Critical questions for big data: Provocations for a cultural, technological, and scholarly phenomenon," *Information, communication & society*, vol. 15, no. 5, pp. 662–679, 2012.

[12] R. Kitchin, "Big data, new epistemologies and paradigm shifts," *Big Data & Society*, vol. 1, no. 1, p. 2053951714528481, 2014.

[13] J. Cleland-Huang, "Don't fire the architect! where were the requirements?" *IEEE software*, vol. 31, no. 2, pp. 27–29, 2014.

[14] J. Oberg, "Why the mars probe went off course," *IEEE Spectrum*, vol. 36, no. 12, pp. 34–39, 1999.

[15] D. Higa, "Walled gardens versus the wild west," *Computer*, vol. 41, no. 10, 2008.

[16] S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010.

[17] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[18] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[19] E. Freeman, E. Robson, B. Bates, and K. Sierra, *Head first design patterns*. O'Reilly Media, Inc., 2004.

[20] P. Gestwicki, "All the uml you need to know." [Online]. Available: http://www.cs.bsu.edu/~pvg/misc/uml/

[21] M. Fowler, *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[22] R. C. Martin, *Agile Principles, Patterns, and Practices in C#*. Prentice Hall, 2006.

[23] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on software engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[24] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[25] G. Booch, "Object-oriented development," *IEEE transactions on Software Engineering*, no. 2, pp. 211–221, 1986.

[26] B. Meyer, "Reusability: The case for object-oriented design," *IEEE software*, vol. 4, no. 2, p. 50, 1987.

[27] L. Hatton, "Does oo sync with how we think?" *IEEE software*, vol. 15, no. 3, pp. 46–54, 1998.

[28] "Iso/iec/ieee approved draft international standard - systems and software engineering – software life cycle processes," *ISO/IEC/IEEE P12207-FDIS-1707, June 2017*, pp. 1–156, Jan 2017.

[29] W. W. Royce, "Managing the development of large software systems," in *proceedings of IEEE WESCON*, vol. 26, no. 8. Los Angeles, 1970, pp. 328–338.

[30] C. Larman and V. R. Basili, "Iterative and incremental developments. a brief history," *Computer*, vol. 36, no. 6, pp. 47–56, June 2003.

[31] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, "lopment methods: Review and analysis," *arXiv preprint arXiv:1709.08439*, 2017.

[32] L. Rising and N. S. Janoff, "The scrum software development process for small teams," *IEEE software*, vol. 17, no. 4, pp. 26–32, 2000.

[33] M. Poppendieck and M. A. Cusumano, "Lean software development: A tutorial," *IEEE software*, vol. 29, no. 5, pp. 26–32, 2012.

[34] A. Cockburn, "Writing effective use cases," 2001.

[35] E. W. Dijkstra, "Letters to the editor: go to statement considered harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968.

[36] F. P. Brooks Jr, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E*. Pearson Education India, 1995.

[37] E. Yourdon, *Death march*. Pearson Education, 2003.

[38] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[39] M. Poppendieck and T. Poppendieck, *Lean software development: an agile toolkit*. Addison-Wesley, 2003.

[40] E. Ries, *The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses*. Crown Books, 2011.

[41] R. Harper, *Practical foundations for programming languages*. Cambridge University Press, 2016.

[42] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[43] M. Paulk, W. Curtis, M. B. Chrissis, and C. Weber, "Capability maturity model for software (version 1.1)," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-93-TR-024, 1993. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11955

[44] J. Highsmith and A. Cockburn, "Agile software development: The business of innovation," *Computer*, vol. 34, no. 9, pp. 120–127, 2001.

[45] T. Dyba and T. Dingsoyr, "What do we know about agile software development?" *IEEE software*, vol. 26, no. 5, pp. 6–9, 2009.

[46] K. Schwaber and J. Sutherland, "The scrum guide," *Scrum Alliance*, vol. 21, 2011.

[47] E. Brechner, *Agile project management with Kanban.* Pearson Education, 2015.

[48] J. Bruner, "The narrative construction of reality," *Critical inquiry*, vol. 18, no. 1, pp. 1–21, 1991.

[49] J. R. Taylor and E. J. Van Every, *The emergent organization: Communication as its site and surface.* Routledge, 1999.

[50] W. Benjamin, "The storyteller: Reflections on the," *The novel: an anthology of criticism and theory, 1900-2000*, p. 361, 2006.

[51] S. Greenblatt, *Marvelous possessions: The wonder of the New World.* OUP Oxford, 1991.

[52] Github. Github git cheat sheet. [Online]. Available: https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf

[53] E. Jendrock, C.-N. Ricardo, I. Evans, K. Hasse, and W. Markito, *The Java EE Tutorial, Release 7.* Oracle, 2014. [Online]. Available: https://docs.oracle.com/javaee/7/JEETT.pdf

[54] A. Tomcat. Tomcat 8.5 user guide. [Online]. Available: http://tomcat.apache.org/tomcat-8.5-doc/index.html

[55] D. Crockford, *JavaScript: The Good Parts: The Good Parts.* " O'Reilly Media, Inc.", 2008.

[56] R. Sedgewick and P. Flajolet, *An introduction to the analysis of algorithms.* Addison-Wesley, 2013.

[57] J. Resig and B. Bibeault, *Secrets of the JavaScript Ninja.* Manning, 2013.

[58] R. T. Fielding and R. N. Taylor, *Architectural styles and the design of network-based software architectures.* University of California, Irvine Doctoral dissertation, 2000.