# Practical Software Engineering for Complex Software Systems

"So you think you can program..."

Steven P. Reiss

Steven P. Reiss
spr@cs.brown.edu

*This page is intentionally left blank.*

# Preface

## 0.1 Preface

preface goes here

# Contents

# Why Software Engineering

# 1. Introduction

This book is about the use of software engineering for complex software systems. In order to understand why these two go together and what is special about the combination, one needs to understand what is software engineering and what are the characteristics that differentiate a complex software system. Moreover, one needs to understand what software engineering has taught us about building complex systems. The lessons learned here are those that are elaborated on throughout the rest of the book. Thus this chapter, along with the next which provides an example system that will be used throughout the book, provide both the motivation and the background the reader will need.

## 1.1 Software Engineering

### 1.1.1 What is Software Engineering

There are lots of definitions available for software engineering. The simplest one is that it is *the application of sound engineering principles to software development*. This definition, however, assumes the reader is well aware of what sound engineering principles are, how engineering is done, and why this matters.

A clearer, but similar, definition is that software engineering is *the application of a systematic and disciplined approach to the development and maintenance of complex software systems*. **[SPR: citation?]** This definition emphasizes that the field has established a set of principles and methodologies that can be used effectively for software development. More importantly, the definition notes that the field has concentrated on complex software systems rather than programming in general. Many of the techniques that have been developed would be overkill if applied to a smaller system and thus many programmers, especially newer programmers, do not appreciate their application in general.

A third definition is that software engineering is *the process of analyzing user requirements and then designing, building, and testing a software system that meets the requirements*. **[SPR: citation?]** This definition is both more specific and limiting. It is more specific in that it notes that software engineering involves phases of development. These phases are similar to those used in traditional engineering, so in that sense the definition lines up with our first. It is limiting in that it only emphasizes development and testing, assumes the requirements are static, and does not cover the long term maintenance of the software.

In a sense all these definitions are correct. Software engineering does look at how to build and maintain complex software systems. It does maintain the notion of phases from engineering. However, it realizes that software is a lot more flexible than a large engineering project such as a bridge which is built once and then rarely changed. Instead, it considers how software evolves over time and how the phases of development are actually intermingled.

### 1.1.2 What has Software Engineering Accomplished

Software engineering has been remarkably successful over the years, even if it this has not been emphasized or touted. The speed of chips doubled every 18 months according to Moore's law, and while this growth has slowed, it is still continuing and still quite impressive. Similarly, the size of storage media has grown by about an order of magnitude every decade. Again, this has been quite impressive. But what about software?

It turns out that what is considered a large, complex software system has increased by an order of magnitude every decade. In the 1970's, most software was developed on punched cards; a box of punched cards held 2,000 cards, and a really large software system was about 10,000 lines. In the 1980's, a 5 megabyte disk was large and a large software system was about 100,000 lines. By the 1990's disks were up to 50 megabytes and a large software system was about a million lines of source. By 2000, a 10 gigabyte disk was available and 10 million line programs existed. In the 2010's one could get a terabyte disk and a large software system was 50 million lines of source (Windows). By 2020, a 10 terabyte disk was standard and a large program was 2 billion lines (google web). **[SPR: Need to include concrete examples here.]** **[SPR: Insert a figure to go along with this. Table showing the year, disk size, memory size, and program size.]**

The techniques that are needed to build larger and larger systems are the result of software engineering, both research and practice. In particular, software engineering has focused on building large, complex systems rather than simple programming. It has developed tools, techniques and frameworks that allow us to build software at scale. And it has identified and addresses the various issues that arise when attempting to build software at these scales. Without software engineering, modern software would not exist.

## 1.2 Complex Software Systems

What makes a particular software system complex so that it requires software engineering? By the standards above, it is the size of the system. But this is only one measure and, although it is indicative, it is not a particularly good one. The size will vary with the programming language used as some languages are more compact or expressive than others. It can vary with the number of comments and blank lines that are inserted or the amount of code that is present but never or rarely used. It also can vary over time as software systems tend to grow as new features are added or bugs are fixed.

A second measure would be the amount of time it takes to build the system. This again is not the best measure. A really good programmer might be ten times more efficient that an average programmer. An experienced programmer will generally be more efficient that a less experienced one. Moreover, if multiple people work on a project, productivity per person tends to decline as the participants need to communicate and understand each other. This phenomena is the ”mythical man-month” of Fred Brooks **[SPR: citation]**, that notes that traditional way of measuring the development time using man-months is not particularly accurate for software,

A third definition would be that a complex software system is one that is too complex for one person to understand. This definition is again vague as different programmers have different abilities to abstract and understand large systems and it is difficult to quantify what understand means.

All these definitions get to the heart of why developing a large software system is difficult. They hint upon the characteristics of complex software that software engineering has addressed. They form the basis for the techniques that are emphasized in this text.

First, large software systems typically require multiple developers. Software engineering has found efficient ways for these developers to collaborate without interfering or hindering each other. Second, large software systems are generally not written all at once. Instead, a smaller version of the system is implemented and that version is then extended with additional features and capabilities. Third, large software systems are usually long-lived. One doesn't write a million lines of code and then throw it away (at least we hope not). These last two characteristics imply that complex systems are not simply written, but rather they evolve over time. Software engineering has developed the methodologies to make this evolution possible and practical. **[SPR: Possibly add a figure that lists the characteristics or problems of complex software systems.]**

Modern software systems have additional complexities that again software engineering has come to address. First, they tend to be distributed systems, running on multiple machines simultaneously. For example, most web applications run in part in the user's browser and in part in the cloud, often with multiple servers handling different parts of the system. Second, modern system tend to make use of the concurrency inherent to today's CPUs, utilizing multiple threads or similar mechanisms.

Finally, as software systems have grown larger, they have become more prone to failure. Most systems are large enough that they are almost guaranteed to have bugs. Modern systems, even production versions of theses systems, can have hundreds or thousands of outstanding bugs. Yet they still work and are still functional. Again, software engineering provides the facilities to both reduce the number of bugs and to recover appropriately without crashing when they occur.

## 1.3 Software Engineering at Scale

Software engineering addresses the problems with developing large scale, long-lived, complex systems.

Software engineering provides the tools and techniques needed to handle large systems. This starts with tools and techniques for handling teams of programmers working on the same system. The techniques attempt to isolate each persons contribution and ensure that it will fit with the rest of the system. The tools provide effective communication and attempt to ensure that the changes made by one developer do not interfere with the changes of another.

Beyond this, software engineering has developed effective frameworks for building distributed and concurrent systems. These range from software architectures that encourage provide well-defined interfaces between components, such as client-server or message based systems, to design patterns and frameworks for avoiding concurrency problems in a system. Software engineering has also provided techniques for preventing and handling failures, both in terms of techniques for testing the software before it is used, and for handling error recovery within the software.

Originally, software engineering realized that the longest part of software development was maintenance of the software. Even so, little emphasis was placed on this aspect of software development. Today, with systems as large and complex as they are, software development is mainly maintenance. While a small system kernel might be built initially, the bulk of a large system is going to be added later. Thus a modern view of software development is that *effectively all software development is maintenance.* Software engineering has come to accept this and to provide appropriate tools and techniques.

Large systems are also long-lived and need to be built and designed accordingly. Today's environments keep changing and long-lived systems need to adapt to these changes. Programming languages and their compilers are updated every year or more frequently. Operating systems are updated every year or two. Mobile devices get new capabilities with every release. New facilities, such as smart computer vision or generative AI based on large language models, become available possibly unexpectedly. User's expect a software system to adopt these new technologies and to work with them as they appear. Thus the ability to evolve a software system has become more important. Software engineering addresses this though both a life cycle model that accommodate evolution and through design techniques that can simplify evolution.

Finally, large, complex software involves risks. Many large software systems fail, either by never working out or by never really being used. Software engineering has taught us the importance of understanding the risks in a software project and of developing the software to manage those risks.

## 1.4 Phases of Software Development

Software development is typically broken up into phases that correspond to the phases used in other forms of engineering. These start with *requirements*, determining what the users needs are. Next comes *specifications* which involves determining what to build to meet these needs. After this comes *design* where the software structure is developed. Then comes *coding* where the code actually written. After coding comes *testing* where one ensures that code works and meets the user's needs. Finally, after testing comes *maintenance* where the software is deployed and then updated.

### 1.4.1 Requirements

The first step in developing a software system is to determine why the system is needed. This involves understanding what problems the software is designed to solve, understanding how it might be used, and understanding how it will interact with existing systems and methodologies. To do this one needs to interact with the potential users of the system. Based on interaction with the users, one determines what is required from the software. This set of requirements then form the basis for determining what software should be written and define the software from the user's point of view.

Requirements are important. Software systems are not always successful. Sometimes they just never work correctly. More often, however, they fail because they solve the wrong problem or solve the problem in a way that is not particularly useful to potential users.

Requirements for large, complex software systems are often difficult to determine. The system will not be ready for some time, possibly years. Moreover, the software should be designed to last and be usable for years to come. Today's users and problems might not be the same as those years from now. Thus, for complex software systems, requirements tend to evolve over time rather than being done all at once. Moreover, because the developers need to track what's happening in the world and adapt the software to this, requirements tend to involve user interaction and user interaction throughout the development process. **[SPR: Reference to chapter 3?]**

### 1.4.2 Specifications

Once developers understand what problem they are going to solve, they can determine a software solution to that problem if one exists. This looks at the requirements again, but instead of looking at it from the user's point of view, it looks at it from the developer's.

Specifications typically define the set of commands and interactions that are required to meet the user's requirements. They take various user requirements and translate these into the actions that the software should take along with the inputs and outputs of those actions. They might provide additional information, for example, sketches of possible user interfaces.

Specifications are useful in several ways. They can be checked against the requirements to ensure that the proposed system meets the user's needs. They can serve as the basis and motivation for design and design decisions. And they can be used to interact with potential users to ensure that the right software system is being built before doing all the work of actually building it.

Specifications for large, complex software systems can be very large. As such, they are often impractical to develop and understand all at once. The approach that is taken today typically involves building a smaller system and then augmenting that system with additional features to achieve a working product. This requires prioritizing the specifications based on the requirements to determine what is in the initial system and what order the various features should be considered in.

Specifications also need to consider the notion of risk. They need to determine what are the portions of the system that might pose a risk to its success and ensure that these are well known to the designer so they can be mitigated during development.

Specifications also need to deal with more than just the functionality of the system. They need to consider the safety, performance, and reliability of the resultant product. They need to consider documentation and maintainability as well, ensuring that users can understand the system and the system can evolve throughout its lifetime. **[SPR: Reference to chapter 4]**

### 1.4.3 Design

Design is typically viewed as the part of software development that requires the most thought and creativity. It is typically done at a variety of levels, from the overall architecture of the system down to designing how an individual function or method should operate. It involves making hundreds or thousands of decisions as to how something should be done. It involves understanding how others have solved similar problems and using those solutions as patterns for the current problem.

As software systems have grown, design has become more hierarchical. Now one can work on the very high level design of the system, its software architecture, to provide an overall framework in which the system will operate effectively. This typically breaks the design into components. Some of these components can be quite large and need to be broken down into smaller pieces. This is high-level design and typically involves thinking in terms of objects. Eventually the design needs to consider how to implement a particular component in terms of classes, methods, and data. This is low-level design.

Design of large, complex systems tends to have more hierarchical levels than the design of smaller systems. It needs to account for risks identified in the specifications and modifying the design to mitigate these risks. It needs to take into account the potential evolution of the system where new features will be added and the system will have to adapt to changing needs and environments. It needs to take into account the development team, ensuring that team members can work somewhat independently to maximize productivity. It needs to take into account the requirements of the user interface needed by the application. And it has to account for the need for concurrency in the resultant system to take advantage of today's processors. **[SPR: reference to chapters 5 thru 9]**

### 1.4.4 Coding

In traditional software engineering, coding, given that one has a detailed design for the task to be coded, is considered the simplest and most straightforward phase of development. In a large system, typically the design is not done down to the finest level of detail, and the actual design of methods, algorithms, support classes, and other implementations are left as part of the coding. This can make coding both challenging and interesting.

More importantly, in a large system how one codes is critical. When there is a programmer team, more than one person needs to be able to read, understand and modify the software. This can involve new team members and can occur after the original developer has left the team. Different team members need to be comfortable handling others code.

Detailed coding also has to account for software evolution. It needs to make it relatively easy to add new features, especially those that were anticipated in the requirements and specifications. It needs to be defensive so that problems introduced during software evolution are caught early and are relatively easy to fix. It needs to account for the complexity of today's user interfaces, that can range from mobile devices to workstations to the web. It needs to take into account security and the identified risks so that there won't be unforeseen problems.

Coding also involves building the software as a team and building it so it is likely to be correct. Modern software engineering has introduced a host of tools to assist in this process, including build tools, team management tools, and fully integrated development environments. Developers should know how to make the most effective use of these tools. **[SPR: reference to chapter 10 thru 15]**

### 1.4.5 Verification and Validation

Once the code is written, it needs to be shown to work. This is done with a combination of verification, proving aspects of the program correct, and validation, testing the program to provide a degree of confidence that it works correctly. Along with these, one needs to think about debugging, both finding problems identified by the testing process and fixing those problems.

The emphasis of the work in this area, especially for large systems, involves testing. Software engineering has developed a variety of testing tools and strategies that can assist developers. Developers should be aware of these tools and should make effective use of them. Testing is an essential part of modern development, with some people advocating that the test cases should be written before the code and that adequate testing is key to successful development.

Verification is more complex. A complex system with a large user interface and many options and interactions with both the user and the real world can be extremely difficult to specify formally, and hence impossible to prove correct. However, it is possible to take particular problems, for example, security problems, abstract those out of the software and prove that the system is correct with respect to that problem and the abstraction. [**SPR: Reference chapters 17 and 18**].

### 1.4.6 Maintenance

Maintenance is the task of keeping the software running over time. This originally involved ensuring it continued working as the underlying software, operating systems, compilers, etc. and hardware evolved. Because of the flexibility of software and the ease relative ease of changing it (say compared to a bridge), maintenance goes beyond fixing any problems that arise and to include adding new features and evolving the software to meet new demands and directions. Maintenance was always seen as the longest phase of software development, but it was one that received little respect.

Large software systems are not developed in one piece. Instead, they are developed over time starting from a core system or from existing code, adding new features, changing user interface, adapting to new situations, and meeting new user requirements. Maintenance still involves keeping the system running as hardware and languages evolve. Today's languages, for example Java, come out with a new version every six months or so and features can disappear in as little as three years. Mobile phones and tablets get new features that users want to be able to use with their applications on a regular basis. AI, especially AI based on large language models, is becoming increasingly powerful and users are beginning to expect it to be used by their applications. Moreover, it is important to keep the evolving application robust and secure. All this means maintenance has become increasingly important.

In addition, users have come to expect the software to change, to add new features, and develop an ”improved” user interface every year or so. Applications need to accommodate these expectations or will lose ground to their competitors. Users also expect the software to be perfect, to do what they think it should do, and to continue working even as their platforms evolve.

The effect of this is that to a first approximation: **All software development of a complex system is maintenance.** This has become a guiding factor in software engineering and in large-scale software development. The result is that software engineering and the material covered in this text has a strong emphasis on making maintenance easy and simple, even at the cost of making some of the other phases of development a bit more complex. For example, one might create a more complex design that allows new features to be added more easily, or one might write code more defensively and with more general interfaces to accommodate future development.

## 1.5 Models of Software Development

Understanding software development means not only understanding the different phases and what they involve, but also understanding how they fit together.

### 1.5.1 Classical Models

Originally, the phases were viewed from a classical engineering background. Creating software was looked at the same way as creating a bridge or a house or a piece of hardware. Here each phase is completed and checked off before the next is begun. The result, when drawn, can look like a waterfall and hence this is called the waterfall model as seen in Figure 1.1.

In the waterfall model, the user first determines what they want. They put together a detailed description, the requirements, and pass them off to the development team. The development team then takes these, analyzes them, and then produces a specification document that describes what the system will do and how the requirements will be met. This document is then approved by the proposers and they are then ready for the next phase. Given the specifications, the developers next do a high level design and then a detailed design, down to the level of individual functions.

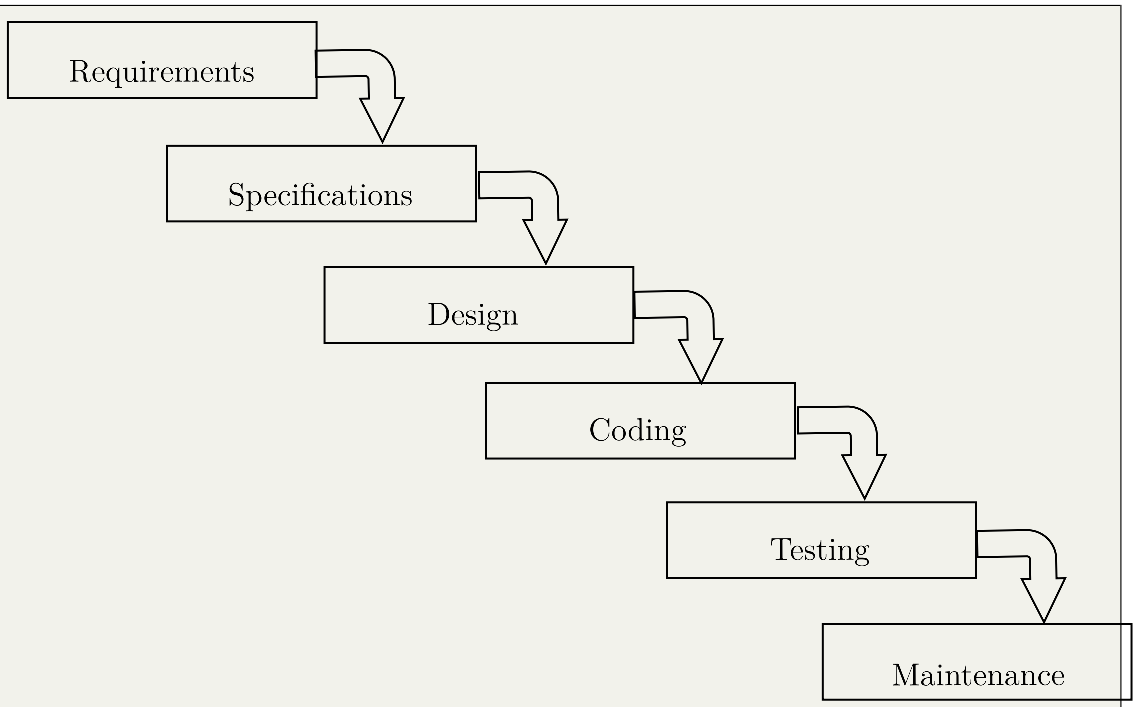


Figure 1.1: Classic Waterfall Model of Software Development **[SPR: redraw]**

This design is reviewed and then approved. Once it is approved, then it is passed to a coding team which actually writes the code. Once the code is written and seems to work, it is passed to a quality analysis team which does testing. Once they approve the system, it is deployed and a maintenance team takes over to keep the product running. Eventually the system will need to be updated in which case the whole process starts again, with users being asked for requirements for the update.

This model is quite simplistic and not particularly realistic given the demands and vagaries of software development. Specifications and the user feedback on them provides opportunities to adjust the requirements. As design progresses, developers find some things easy, even if they weren't in the requirements, and other things that might be in the requirements but unimportant very difficult, allowing the specifications to be changed. As implementation proceeds, one finds design problems that were overlooked and a modified design needs to be created. Testing identifies problems that require code modifications. And maintenance can require additional

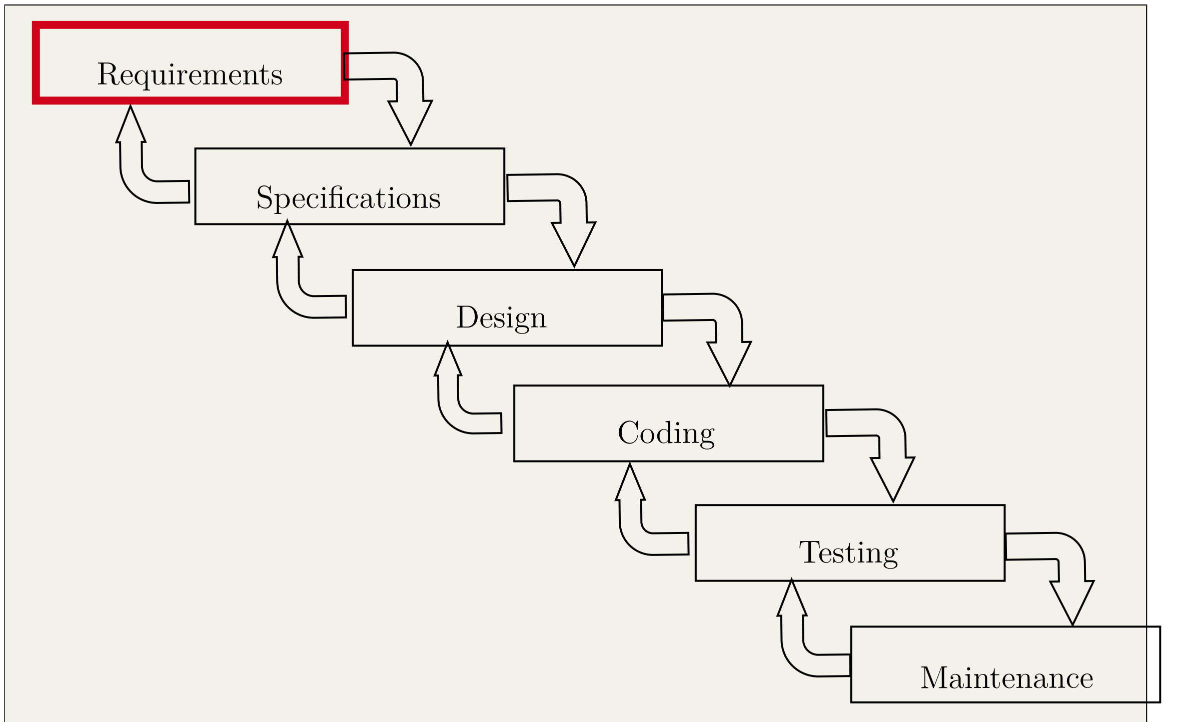


Figure 1.2: Feedback Waterfall Model **[SPR: redraw]**

requirements, specifications, design, coding, and testing. A more realistic approach is a model that includes such feedback as optional paths added to the waterfall model. This is the feedback waterfall model as seen in Figure 1.2.

The feedback waterfall model still assumes that most software development is done in one shot, with initial design reflecting the complete specifications and requirements. As systems have grown larger, this is impractical. Instead the system is developed by building a core system, then augmenting that system with additional features in stages. Each stage involves its own requirements, specifications, coding and testing. This is best reflected in the spiral model proposed by Barry Boehm **[SPR: Citation]** as seen in Figure 1.3. This model involves viewing the initial implementations of the system as prototypes to test concepts and get feedback from users. These prototypes are then used as the basis for designing and building the final system.

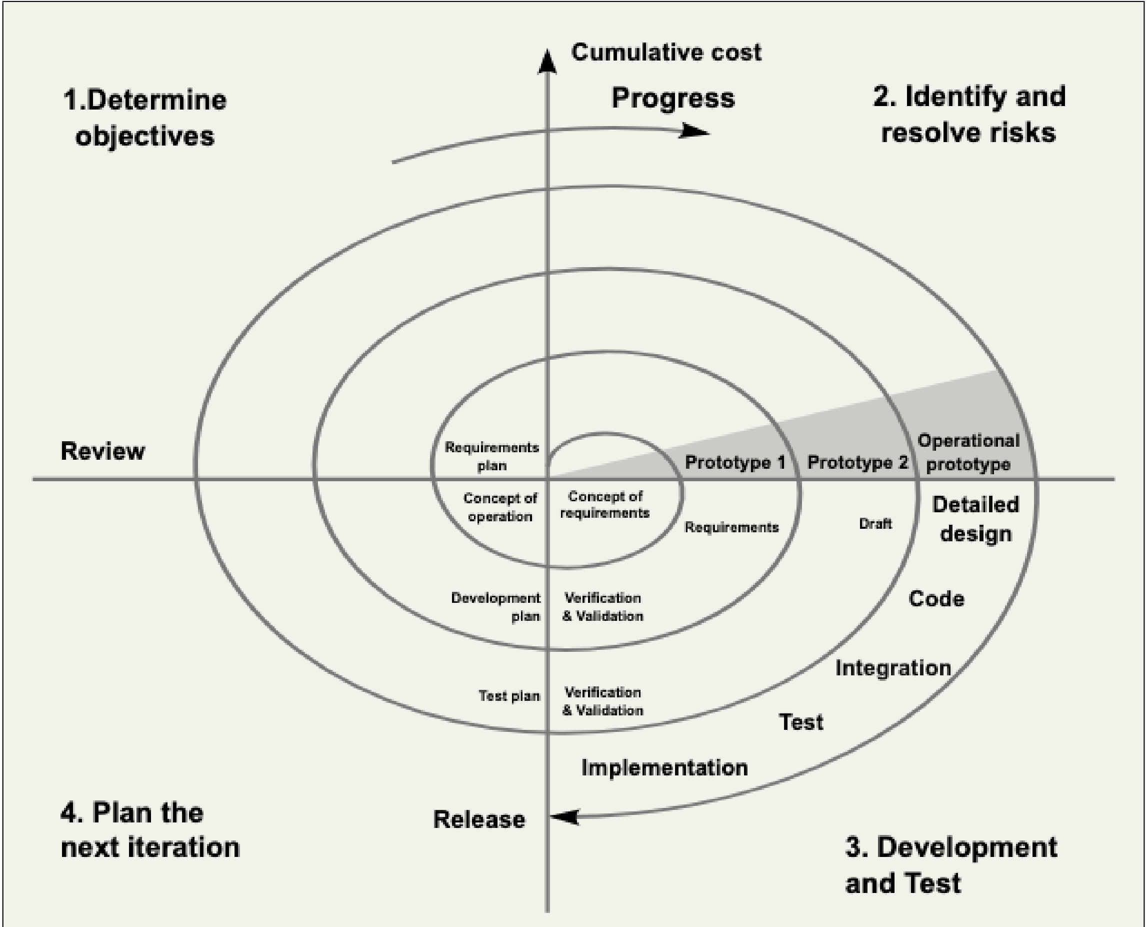


Figure 1.3: Spiral Model [SPR: replace with our own diagram]

### 1.5.2 Agile Models

As systems continued to grow it was realized that developers want to have a working system as early as possible and continually add features. Rather that spiral model with a small number of generated prototypes, developers wanted to build and deploy an actual system and then add features based on requirements and feedback from users. This is the basis for agile development.

Agile development is more of a circular model. It starts with planning the system based on some initial set of specifications. This is designed, developed, tested, and then deployed. After deployment, the system is reviewed, often based on feedback

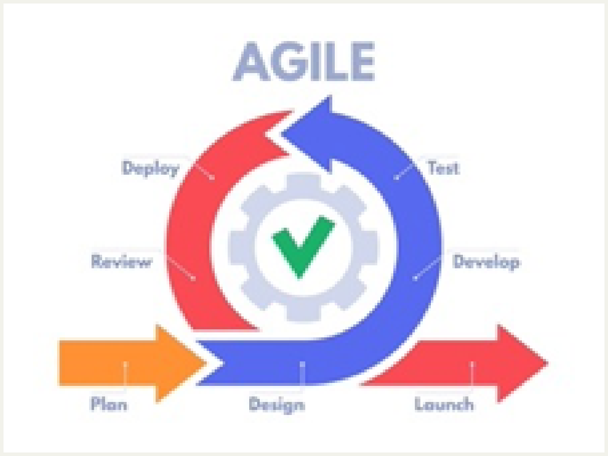


Figure 1.4: Agile model **[SPR: replace with our own diagram]**

from test users, and, and a set of features to be added is decided upon. These new features are then designed, developed, tested, and deployed and then the process starts over again. Eventually the system is complete and the product is launched. This is reflected in a circular model of software development seen in Figure 1.4.

Agile development evolved from a movement called Extreme Programming. This espoused a number of other techniques including pair programming; test-first development where the test cases are written before the code is developed; relatively frequent code refactoring; and rapid turnaround on adding features to the system. Pair programming and test-first development can be integrated into any of these models of software development as part of coding. Refactoring implies a willingness to redesign the software as needed, often requiring a major rewrite of parts of the system.

Rapid turnaround has been widely adapted as part of agile development. A set of features to be added is viewed as a sprint. Typically sprints are done in a one or two week time frame. The programming team meets at least weekly to discuss where they are and to consider the next sprint. Scrum development expands on this by having more frequent meetings, often daily, where the developers come to understand the current state of development and reorient the current sprint as needed.

### 1.5.3 A Compromise Model

While agile development is a step in the right direction, it does not capture the emphasis on maintenance that is the reality of modern software development. We prefer a model that embodies continuous development while at the same time attempts to anticipate future problems and simplify the addition of new features.

In particular, agile development looks only at the current needs of the system and emphasizes adding new features one at a time. It assumes that if a new feature does not fit nicely with the current system framework, that the system will be refactored accordingly. This means that the initial system can be simpler since it does not have to account for lots of extensions that may or may not be needed. It puts the cost of adding these extensions off, making initial development less expensive, but at the cost of requiring significant refactoring and cost latter in development.

Our experience is that it is relatively easy and inexpensive in terms of coding and design to anticipate possible future changes early in the development process. It is possible to design the system so that a portions of the system that are likely to change can be identified. It is also possible to note what features might be added and design the system so that adding these features would be relatively easy. The modifications needed to do this are generally pretty simple and often result in a better, more general and more flexible design for the overall system. Moreover the cost of designing the eventual system this way is often much less than attempting to do the necessary refactorings after a portion of the system has been built.

Based on this, we prefer a model that starts by considering requirements and specifications for the a full system, much like the water fall model, but with considerable feedback from users on both the requirements and specifications. The requirements and specifications phases have a lot in common, but are done from different perspectives, requirements from that of the user, specifications from that of the developer. Requirements lead to specifications. Specifications then provide the user with a better understanding of what can be done and yield new or revised requirements. What should be done in practice is to develop requirements and specifications together, using one to drive the other and using feedback from both users and developers to create a more practical and viable view of the system for design purposes. The combination of the two is called *requirement specifications*, emphasizing that the two phases are closely tied together.

A compromise model does requirement specifications for a complete, fully-featured system. While changing system and user requirements might make some of the spec-

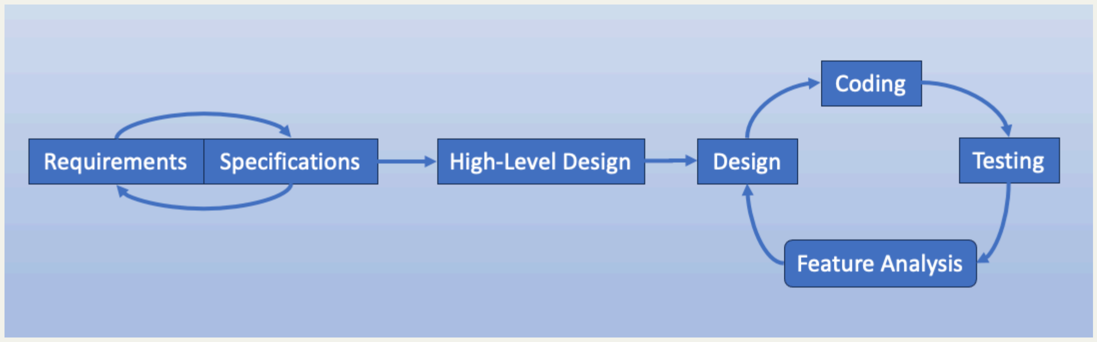


Figure 1.5: Lollipop Model

ified features irrelevant, understanding the possible directions that the system might take still yields a better design and helps to eliminate or simplify future refactoring.

The full set of specifications is used for two things. First, it is used as the basis for choosing an appropriate high level architecture and design for the resultant software system. This software architecture defines the primary components of the system and how they interact with each other. Having an appropriate architecture provides a framework for building the remainder of the system. **[SPR: Reference to chapter 5]** The high level design describes the components in detail so they can be understood and eventually implemented.

Second, the full specifications are used as the basis for developing the product. The specifications are prioritized and a the minimal set needed for a viable product is be defined. This minimal product is the first thing that is designed, built tested and deployed, often with the core functionality designed and built first. However, the design and coding both attempt to take into account the possible future features based on the full specifications.

After the minimal system is available, the next set of features, based on the initial priorities and any feedback from users of the product is decided upon. These are used in the next cycle of development starting with specifications for these features and ending with deployment. This process is continuous. It does not anticipate stopping at some point, but rather assumes one is continually adding or modifying features of the software, trying to develop the next version of the system. We call this a lollipop model as seen in Figure 1.5. **[SPR: balloon model? both are based on trails with path going out to a loop.]**

## 1.6 Software Tools

A large part of software engineering research and effort has been devoted to developing tools to help developers design and build better, higher-quality software more efficiently. Tools have been developed for all phases of software development, with differing degrees of success.

Tools for requirements and specifications provide a means of organizing, accessing, tracking, and linking the results of these phases to each other and to the rest of the code. They are most useful when the specifications are complex and mainly developed in advance. They are most widely used in larger companies on well-defined projects.

Tools for design have attempted to provide design representations that are easy to create, understand and manipulate. Much of the work in this area culminated in the unified modeling language (UML) and tools that support graphically creating UML-based designs. While UML was widely touted, its use today is more limited than was originally envisioned. Still, UML diagrams provide a universal design language, have a variety of uses in software development, and should be known by developers.

Tools for coding are the most common. The most widely used tools are integrated development environments (IDEs) such as VS Code, Eclipse, IntelliJ or our own Code Bubbles. These have been under development since the 1980s and have matured to the point where there use is *de facto* for developers. These IDEs support editing, navigation, code exploration, and debugging. They also provide interfaces to other development tools. These other development tools include facilities for team communication and organization such as Slack, project managers, tools for managing source code versions and history such as Git, and tools for building complex software systems such as Maven, Gradle and Ant.

Beyond coding, there are tools that support testing, both of the code, for example Junit for Java, and of user interfaces, for example Selenium. Keeping track of the current bugs in the system and their corresponding test cases and status is the job of bug repositories. Finally, there are tools that assist system deployment, for example Docker and installation managers.

This book will discuss these tools in the appropriate sections. Developers working on their own project should use their own selection of tools, often those dictated by their company or school, but also dependent on the development language or languages, tool availability, previous experience, and what seems most appropriate for their particular development situation.

**[SPR: Need to establish consistent terminology for use throughout the text. In particular:**

- ***application, system, target.*** **Need a consistent name for what is being built.**
- ***long-lived, complex, large software.*** **Need a consistent name for the type of software we are aiming at. This should be introduced in the appropriate section in this chapter.**
- ***programmer, developer, designer.*** **Need a consistent way of referring to the people doing the development, the target audience of the text.**
- ***person.*** **Are we addressing the reader from the 1st or 3rd person. Are we addressing them as 2nd person or 3rd person. Should be consistent throughout.**

]

## 1.7 Summary

**[SPR: Emphasize that all software development is maintenance and that all the various phases of software development should be geared to making maintenance easier. Note we will continually come back to this theme as we go through the text.]**

## 1.8 Further Reading

References for this chapter? Or provide a reference to a more detailed discussion of software development life cycle models and a reference to agile development.

# 2. A Sample Problem

The best way of understanding how to successfully build complex software systems is by building them. The second best is to use an existing system as an example and understand how it was developed and what made that development work and why.

In this chapter we consider a smallish software system (under 20,000 lines of source) that will be used to illustrate the various principles and techniques described in this book. This chapter provides the motivation and a high level description of the goals of that system. As we get further into the text, we will look at detail requirements and specifications, designs, code, testing and maintenance of the system.

One example however is not enough to convey all the principles of applied software engineering. This chapter provides an overview of a number of other software systems developed by the author that are used as examples, sometimes as examples of what to do, sometimes as examples of what not to do.

## 2.1 SHORE: Controlling a Model Railroad

I have a HO railroad set up that I put together 25-30 years ago. Its somewhat complex, sitting on a 12 foot by 4 foot table, and can run two trains independently on the mainline and have one or two more operating in a rail yard complete with sidings and a turntable. There are 13 rail switches (Y's) (with 11 controls since there are two pairs that operate together). There is a crossover and a reversing track on the main line. And there are 14 track blocks, sections of track that have independent power controls.

Needless to say, this is rather complex to operate, especially for one person. In addition to the 11 buttons to control the 13 switches, there are 14 power controls for the different blocks, 3 rheostats to control speed, 4 forward-off-reverse controls, and

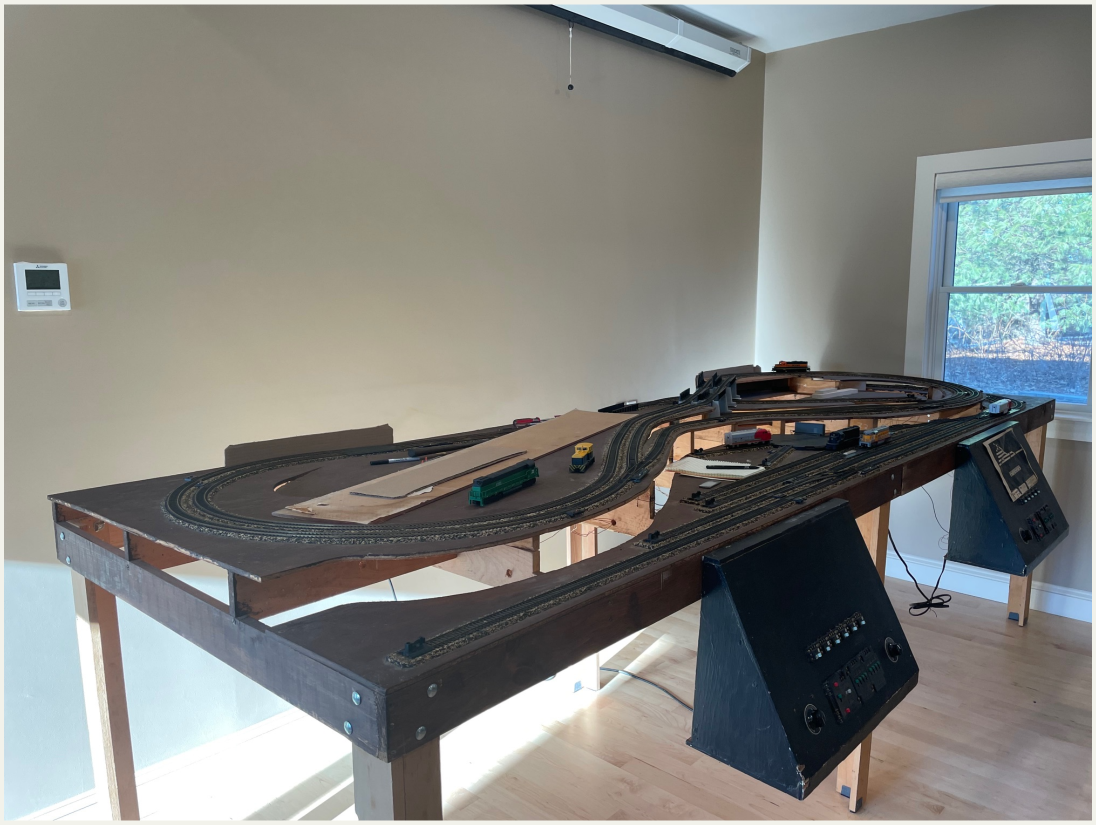


Figure 2.1: Our HO Train Table

a control for the turntable. These are spread over two control panels. This can be seen in Figure 2.1. **[SPR: need a better picture]**

One has to prevent trains from both derailing and crashing. Derailing occurs most commonly when a train tries to pass over a switch that is set incorrectly. It can also occur when a train is moving to fast around a curve or over a bridge. Collisions can occur when two trains converge on a switch or if two trains are on the same track and one is going faster than the other. The standard approach to preventing collisions is to ensure that only one train at a time is in a track block.

I want to control this setup from my computer. Moreover, I want to have a high level control so I can say things like ”Take train A out of the yard, go around the outer loop 3 times, then around the inner loop twice, then reverse directions and go around the outer loop twice more before bringing the train back into the yard." At the same time, I want to ensure that the trains do not crash or derail. This is the system that should be written.

### 2.1.1 Underlying Hardware

In order to have computer control of the HO layout, one needs to enable the computer to both understand the current conditions, for example, what state switches are in and where trains are, and to control things.

Our layout, because it is relatively old, uses all manual controls. These include rheostats, buttons for each switch, and controls to select the power source and direction for each track block. Shortly after the layout was built, trains started to support Digital Command Control (DCC). Here there is continuous power to all rails and the control is in the engine. DCC sends signals to the engines via pulses on top of the rail power. This is done using special controllers that are attached to the circuit. This allows each engine to independently go forwards or backwards and to controls speed. This type of control is much simpler and closer to the real world.

DCC, however, is starting to fall out of favor. Today one can find the equivalent controls using WiFi so that the engines can be controlled from ones phone or tablet and no special hardware is required. While DCC has been standardized, WiFi control has not.

While it is possible to create a circuit that would effectively actuate the original set of controls for the HO layout, we decided that it made more sense to upgrade the engines to use the latest technology. Again, while we could do this via DCC (there are Arduino-based DCC controllers), we decided that we wanted to go with WiFi and use the LocoFi implementation **[SPR: citation]**.

This means that the system needs to control the switches and detect of the state of the trains. We inserted photo diodes in the tracks at key locations. These include on each side the gap between track blocks to detect when a train moves from one block to another; on entry to the two inputs to a switch to ensure the switch is set correctly before the train goes over it; and at locations where we might want a train to stop (red signal) before entering a new block.

To go along with the LocoFi engine control, we designed and built a set of track controllers. Each of these can control 3 switches, handle 8 sensors (photo-diodes), and set 6 signals. An additional controller can handle up to 40 sensors. These communicate using WiFi using a protocol compatible with the LocoFi protocol. These are all

Arduino-based. **[SPR: Cite and give location of circuit schematic, layout, and Arduino program.]**

### 2.1.2 Software Overview

The software we envision would allow both manual and automatic control of the complete layout. The user interface would display the current state of the switches, signals, and trains as the system knows them. It would let the user control the switches and signals directly. It would provide control of the engines in a manner similar to that offered by the LocoFi phone app.

The standard way of displaying a train layout, both in model railroading and for real trains, is to have an display an abstraction of the actual layout. This has a one-to-one mapping with the actual layout, but may omit some details such as curves. For a complex layout, there can be multiple such abstract displays that have connections from one to another. This can be seen in our initial system where the two control panels each have their own abstraction. For example, the abstract layout or our HO trains is shown in two diagrams in Figure 2.2. The software we envision should provide such an abstraction as part of the interface.

The software would also try to ensure the stability of the system. It would prevent two trains from being in the same block at the same time. It would ensure that a switch was set property when a train was passing over it. It would ensure that signals are set correctly and that trains obeyed the signals.

The software would also provide the type of high level control that we envision where one could define a route for a train and have the train correctly follow that route while obeying the various safety conditions.

The system should also be flexible. The embedded sensors do not provide complete information about train position. They have problems understanding the initial conditions, can miss trains, especially in low light, and there are significant spans of track with no sensors. The system should allow us to use a video camera aimed at the track to obtain better information about the current state of trains. Note this isn't perfect either since there might be portions of the track, for example, tunnels, that would not be visible to the camera.

The system should also be flexible in that it should be able to handle layouts beyond ours. We do not include examples of all types of tracks and conditions. For example, we only have 2-way switches while 3-way switches exist, and we have no X-crossings which induce other types of anti-collision constraints.

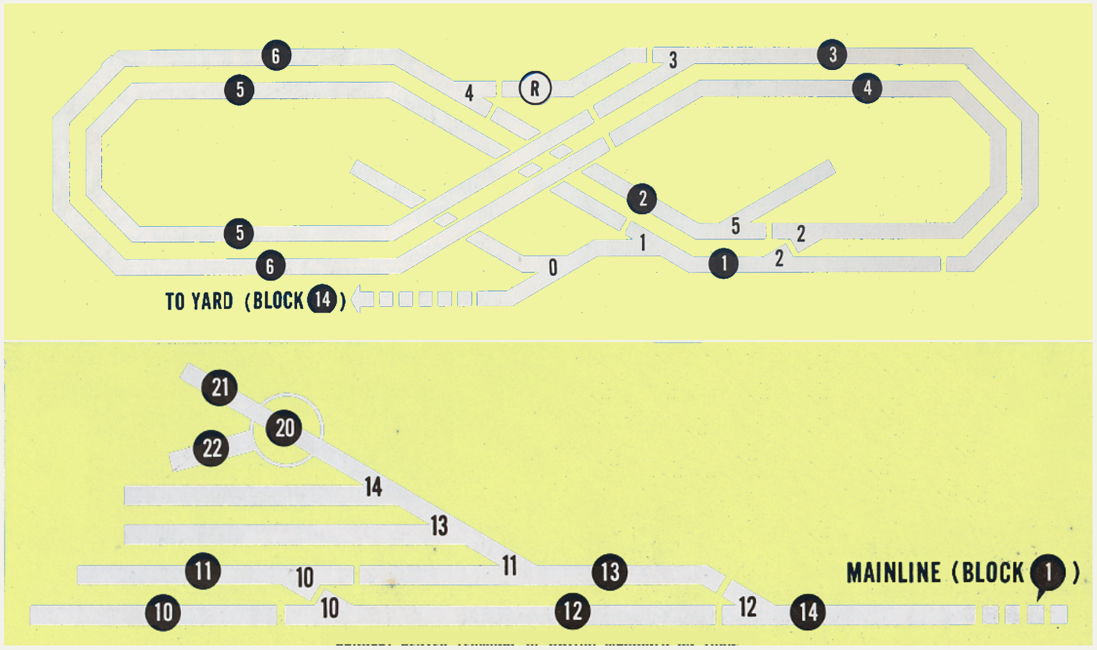


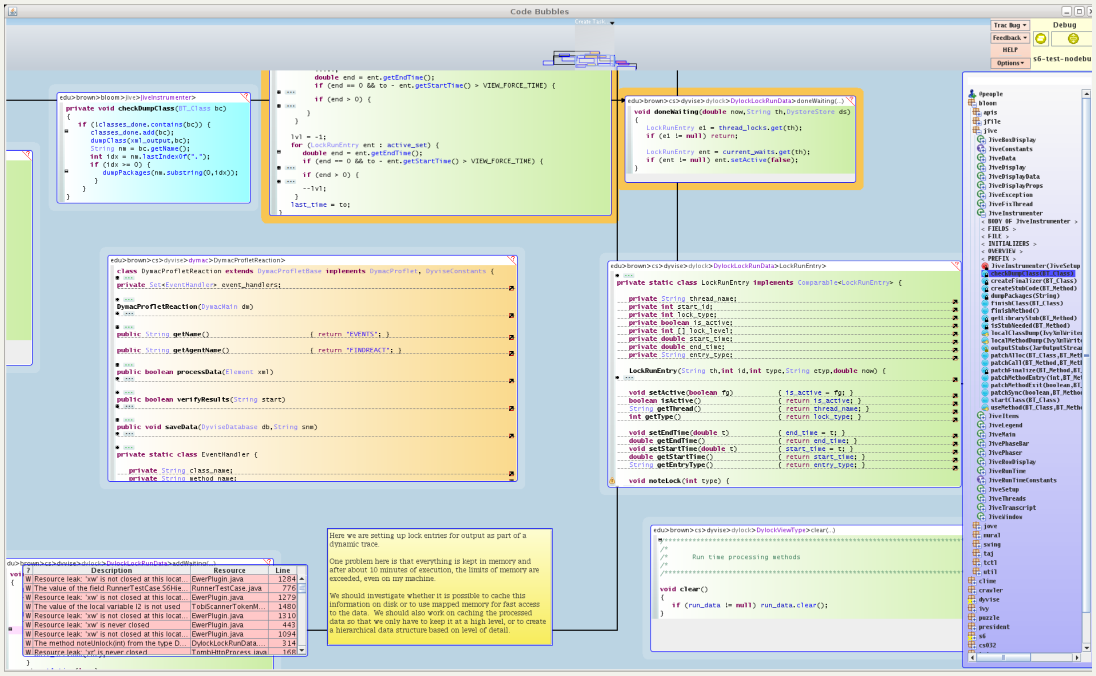
Figure 2.2: Abstract Layout of our HO Trains

As we proceed through the text, we will use this system as an illustration of how to build complex software systems. We will develop requirements and specifications for it. We will choose a software architecture and then do a top-level and portions of the detailed design for the system. We will show samples of the actual code and test cases. We have called the system SHORE for Smart HO Railroad Environment.

## 2.2 Other Systems

One system, while it can provide good illustrations of how modern software development should be done, can never provide a full set of examples.

We have been developing and working with software systems for over 50 years. These systems have taught us what works and what does not. They have provided lessons in what to do and what not to do while developing software. Where appropriate, when the model railroad example is insufficient, this book uses examples drawn from our experience. In this section we provide a brief overview of these systems. Additional relevant details are provided with the examples.



edu>brown>bloom>jive>JiveInstrumenter>
```
private void checkDumpClass(BT_Class bc)
{
   if (!classes_done.contains(bc)) {
      classes_done.add(bc);
      dumpClass(xml_output,bc);
      String nm = bc.getName();
      int idx = nm.lastIndexOf(".");
      if (idx >= 0) {
         dumpPackages(nm.substring(0,idx));
       }
    }
}
```

```
         double end = ent.getEndTime();
         if (end == 0 && to - ent.getStartTime() > VIEW_FORCE_TIME) {
 ...
         if (end > 0) {
 ...
       }
    }

   lvl = -1;
   for (LockRunEntry ent : active_set) {
      double end = ent.getEndTime();
      if (end == 0 && to - ent.getStartTime() > VIEW_FORCE_TIME) {
 ...
      if (end > 0) {
 ...
      --lvl;
    }
   last_time = to;
}
```

edu>brown>cs>dyvise>dylock>DylockLockRunData>doneWaiting(...)
```
void doneWaiting(double now,String th,DystoreStore ds)
{
   LockRunEntry e1 = thread_locks.get(th);
   if (e1 != null) return;

   LockRunEntry ent = current_waits.get(th);
   if (ent != null) ent.setActive(false);
}
```

edu>brown>cs>dyvise>dymac>DymacProfletReaction>
```
class DymacProfletReaction extends DymacProfletBase implements DymacProflet, DyviseConstants {
 ...
private Set<EventHandler> event_handlers;

 ...
DymacProfletReaction(DymacMain dm)
 ...

 ...
public String getName()                          { return "EVENTS"; }

public String getAgentName()                     { return "FINDREACT"; }

 ...
public boolean processData(Element xml)
 ...

 ...
public boolean verifyResults(String start)
 ...

 ...
public void saveData(DyviseDatabase db,String snm)
 ...

 ...
private static class EventHandler {

   private String class_name;
   private String method_name;
```

edu>brown>cs>dyvise>dylock>DylockLockRunData>LockRunEntry>
```
 ...
private static class LockRunEntry implements Comparable<LockRunEntry> {

   private String thread_name;
   private int start_id;
   private int lock_type;
   private boolean is_active;
   private int [] lock_level;
   private double start_time;
   private double end_time;
   private String entry_type;

   LockRunEntry(String th,int id,int type,String etyp,double now) {
 ...

   void setActive(boolean fg)            { is_active = fg; }
   boolean isActive()                    { return is_active; }
   String getThread()                    { return thread_name; }
   int getType()                         { return lock_type; }

   void setEndTime(double t)             { end_time = t; }
   double getEndTime()                   { return end_time; }
   void setStartTime(double t)           { start_time = t; }
   double getStartTime()                 { return start_time; }
   String getEntryType()                 { return entry_type; }

   void noteLock(int type) {
```

du>brown>cs>dyvise>dylock>DylockLockRunData>addWaiting(...)

| ? | Description | Resource | Line |
|---|---|---|---|
| W | Resource leak: 'xw' is not closed at this locat... | EwerPlugin.java | 1284 |
| W | The value of the field RunnerTestCase.S6Hie... | RunnerTestCase.java | 776 |
| W | Resource leak: 'xw' is not closed at this locat... | EwerPlugin.java | 1279 |
| W | The value of the local variable I2 is not used | TobiScannerTokenM... | 1480 |
| W | Resource leak: 'xw' is not closed at this locat... | EwerPlugin.java | 1310 |
| W | Resource leak: 'xw' is never closed | EwerPlugin.java | 443 |
| W | Resource leak: 'xw' is not closed at this locat... | EwerPlugin.java | 1094 |
| W | The method noteUnlock(int) from the type D... | DylockLockRunData.... | 314 |
| W | Resource leak: 'xr' is never closed | TombHttpProcess.java | 168 |

Here we are setting up lock entries for output as part of a dynamic trace.

One problem here is that everything is kept in memory and after about 10 minutes of execution, the limits of memory are exceeded, even on my machine.

We should investigate whether it is possible to cache this information on disk or to use mapped memory for fast access to the data. We should also work on caching the processed data so that we only have to keep it at a high level, or to create a hierarchical data structure based on level of detail.

edu>brown>cs>dyvise>dylock>DylockViewType>clear(...)
```
/************************************************************
/*
/*      Run time processing methods
/*
/************************************************************

void clear()
{
   if (run_data != null) run_data.clear();
}
```

Figure 2.3: Screenshot of Code Bubbles

## 2.2.1 Code Bubbles

Code Bubbles is an integrated development environment (IDE) that was originally developed in 2010 **[SPR: citation]** and has been maintained and extended ever since. It is used primarily for Java, but now has become available for other languages as well, notable JavaScript with Node.js and dart/flutter.

Code Bubbles is a working-set based IDE. It lets the developer display and work on their whole working set at once. It provides a variety of techniques to simplify navigation. The net effect is that navigation within the IDE, which can take up to 50% of the developer's time, is virtually eliminated and developers are much more productive. A view of the Code Bubbles environment can be seen in Figure 2.3.

Code Bubbles is a rather complex system, especially with the newer additions that have been included in it. The original implementation is mainly a front end for program development, relying on the Eclipse IDE to handle back-end tasks such as compilation, search, and interaction with the debugger. However, the environment is a complete IDE, providing facilities for search, testing, debugging, visualization

**[SPR: citation]**, documentation, both code and text editing, collaboration, source code management, and repository code search **[SPR: citation]**. More recent plug-ins extend the original system with facilities for live programming in Java **[SPR: citation]**, continually checking of safety conditions through flow analysis **[SPR: citation]**, and automatic program repair while debugging **[SPR: citation]**.

**[SPR: Do we need to go into more detail at this point. Probably can defer the details, say of the architecture, code, size, etc. until later. Might want to note that the system with all the plugins is about 500K LOS and it is written mainly in Java. ]**

Code Bubbles is open source and freely available. **[SPR: Github link and web page here.]** This text will use it when we need an example of a more complex system and how systems evolve over time.

### 2.2.2 Other Systems

Add new subsections here as we cite other systems later in the text. **[SPR: Need to determine what systems are needed here. Possible FAIT, SEEDE, SHERPA, UPOD, TWITTERDATA. Note that the simple ones, such as twitterdata can be described in place when needed, especially if only used once.]**

## 2.3 Summary

Simple summary? Importance of previous knowledge on what works and what doesn't? Importance of examples?

## 2.4 Further Reading

References for this chapter?

## 2.5 Exercises

Introduce a small project such as bouncing balls and have the user think about what this would mean and what they would want. We should have suggestions for 2-3 such small but open-ended projects that readers can use to follow along and play with as they go through the text.

The text can also be used while doing development on a course-sized (or larger) team project. This project should be up to the user (or assigned by the professor or their management). Exercises on this should involve thinking about the system that you want to or plan to build. Provide a description of that system and what you want to accomplish with it.

# Requirements Specifications

# 3. Requirements

The first step in developing a software system is understanding what problem that system is being designed to solve. This involves understanding what potential users do, what problems exist with their current solutions, what new solutions would be better, and what these users would want out of a new system.

## 3.1 What are Requirements

Requirement analysis is the process of establishing the needs of stakeholders to be solved by the system. It generates a definition of the problem to be solved, not a description of its solution. This has to be kept in mind while developing requirements. If one focuses on a particular solution without first understanding the problem, then the particular solution will be the result, but that result is likely not to address the actual problem.

While one generally talks about requirements from the user's point of view, the above definition mentions *stakeholders*. Typically, there are more people interested in a solution than just those who are going to use it. For example, in a company, while lower-level people might use the software, managers are concerned with what they do, how efficient they are, and possibly what information they can gather from the software that can help them in management. Others in the company might be interested in what the software will cost, both initially and in the longer-term. If the software is being used for interacting with other people, say the software is used by customer service agents to deal with customers, then the final customers can be considered stakeholders. In general it can be difficult, especially for developers, to determine the complete set of stakeholders for a system.

Requirement then define a problem and outline possible solutions from the perspective of the stakeholders. They do not define a particular solution, but rather provide constraints on the eventual solution. They indicate what aspects of a solution are important to the stackholders and detail those portions of the problem the solution should focus on to maximize the benefits of the target system. They might define how and where the solution should work, and what existing or other technologies the solution needs to work with. They provide the basis for finding an appropriate solution, which will be done in the next stage, specifications.

Requirements are essential to the success of a software system. Many software systems get built but are never used or are used for a short time and then fade away. These are software failures. Previous studies of software failures have shown that a significant portion of these failures are due to poor requirements. **[SPR: citation]** About a quarter of software failures are due to incomplete requirements. Incomplete requirements result in building a system that does not met users needs or solves the wrong problem. Another quarter are due to the lack of stakeholder involvement in developing specifications. Guessing what the stakeholders would want and use can be haphazard. Hearing from actual users and other stakeholders provides a more solid basis for building the correct system. About a fifth of failures are due to unrealistic expectations, either on the part of the users or by the developers who might encourage users to expect more than they can provide. Changing requirements and changing environments, especially environments where the problem goes away before the solution can be delivered, account for most of the rest of the failures.

## 3.2 The Requirements Process

Requirements analysis is an on-going process. It is the first step in determining what software system to build. But it is also something that continues as the software is being built and as the software evolves. You should think of it as a dialog between the developers and the stakeholders that continues throughout the lifetime of the software.

Requirement analysis is on-going because things are going to change. People's needs will change with time. What the stakeholders need today is not necessarily what they need next month or next year. Markets change. Competition might make your software look or feel obsolete and your stakeholders will want more. New opportunities will arise. While developing the software you might determine that is would be relatively simple to add a new feature. You then should determine if that feature might be of interest to the stakeholder. Stakeholders, seeing the actual or proposed system, might have ideas on how they would want to use it that they did not think of before.

During conversations with the stakeholders, you should keep the goals of requirements in mind. You should try to avoid biasing your conversations by assuming a particular solution of software system. Your primary goal is to define the problem from the user's or stakeholder's point of view. You need to assess the need for a solution, determining if a problem even exists that can be addressed by software. You should determine the outlines of what the stakeholders would see as the ideal solution.

You also have to be practical about what can and cannot be done. You need to prioritize the various user needs and requests. You should determine which of their desires are required as part of the solution and which are actually optional or can come later. You should determine how certain they are of the various needs. You should assess what resource limitations will be imposed on both the running software and on the development. Finally, if you are selling the software or if you expect users to adapt the software once it exists, you should determine acceptance criteria. This can be what features and capabilities need to be present for them to pay for the software or it can be what is the minimal system the user will need before they put in the time and effort to learn and use your new software. **[SPR: Figure listing goals of requirements?]**

Because requirements can and do change over time, it is important in a project to keep track of the current set of requirements. A number of tools have been developed for tracking and representing requirements. Most of these are oriented to very large systems developed for a particular customer where the customer develops the requirements in advance of any software development. They make it easy to add and edit requirements. They ensure that requirements are always available to those who need them. They also provide facilities for traceability, tracking how and where the various requirements affect the latter specifications and design. These include JAMA **[SPR: citation]**, IBM Doors **[SPR: citation]**, and CaseComplete **[SPR: citation]**. **[SPR: Make sure these still exists.]**

While these tools are often overkill for the more common medium size software project, especially one that is begin developed in conjunction with the stakeholders where the requirements are evolving along with the system, they can still be useful. Simpler approaches are also available. One can use a word processor such as Microsoft Word or Google Docs to build the requirements as an outline document. One can also use the facilities of UML to create use cases that represent the requirements. This should be clearer in the next section.

## 3.3 Representing Requirements

There are a two primary methods for representing requirements. The first is to use natural language and organize the requirements in a hierarchical fashion. The second, developed as part of agile programming, is to use use cases to express what users expect from a system. In practice a combination of the two provide a clearer understanding of what the users expect and a better understanding of what the solution should entail and what constraints it needs to meet.

### 3.3.1 List-Based Requirements

The classical way of representing requirements is as a hierarchical, sectioned, and numbered list, much like an outline. The basic sections in this case represent the different functionalities that the user needs from the software. Latter sections can represent other, non-functional constraints on the resultant system that are imposed by the various stakeholders.

The detailed requirements are the leafs of this list. They contain a formalized natural language description of what the user expects out of the system. We say formalized because they need to be quite precise and not in any way ambiguous. Each requirement is numbered, using its hierarchical position, for example I.A.3. Requirements can be duplicated, at least in part, in multiple sections of the document. In this case, the duplicated sections should refer to one another, using their numbering.

This type of requirements provide a formal definition of what is required. It is used, for example, in government contracts. As such, it provides a criteria upon which the resultant system can be measured and can be used to determine when the system meets the needs and when the contractee will pay the developers.

Its best feature is that it provides a basis for requirements traceability. Requirements traceability tries to ensure that the specifications and then later the design and the eventual system are well motivated by the requirements and satisfy the various stakeholders needs. This can be done by adding to each specification what set of requirements it is based on using the numbers of those requirements. For design, it can be used to document why a particular feature is included and why the design is the way it is. Traceability also is used to ensure that all the items specified in the requirements are actually reflected in the specifications and then in the final system. Having a precise, detailed, and relatively complete set of requirements lets the developer continually check the status of software development and to ensure the system keeps on track.

List-based requirements, while they can provide a semi-formal definition of what is required and can provide traceability, are difficult for both the stakeholders and for developers. For stakeholders, it can be difficult to express how they want to use the system since each aspect needs to be sorted into the particular section and subsection. A particular need might have to be in the requirements document multiple times in multiple places and it is relatively easy to overlook one or several of these.

It is also relatively easy to create requirements that are contradictory, with one requirement stating one thing and a second stating its opposite, especially if multiple stakeholders are involved. A list-based requirements document is also difficult to read and does not provide a good overview of the system. Instead the overview needs to be reconstructed by the reader from the low-level details. Incomplete requirements are another problem as it can be difficult to determine if everything that needs to be covered is covered by some item in the document.

For the same reason, list-based requirements are difficult for the developer. In order to be useful for specifications and then design, the developer needs to have a complete understanding of all the requirements and then use that to put together their model of what the stakeholders actually want and how it should work. Since the stakeholders and developers often come from different domains and different perspectives, the understanding developers create this way can be a mismatch to the understanding the stakeholders had in mind.

**[SPR: Can we get a snapshot of a formal requirements document that is public as a figure here or should we defer (and possibly forward reference) to the SHORE requirements section?]**

### 3.3.2 Agile Requirements

Agile development assumes that developers start with a small core system and add one feature at a time. It looks at requirements locally so that the requirements are based on that one feature and not the complete system. Initial requirements are designed to give the stakeholders and developers a feel for how the system would be used. Each round of development is then fed back to the stakeholders for their input and for further requirements.

This is all based on the assumption that it is difficult to develop correct requirements without seeing or having some experience with the system. Indeed, in real world situations, this is often the case. This is why one views requirements as a process rather than a single step.

Rather than attempting to provide a formal definition of the problem, the agile methodology tries to convey from the stakeholders to the developers how the system would be used, what they want the system to do for them, and what the system should and should not do. This information is conveyed in the form of use cases, stories, or scenarios. These are done incrementally, with new use cases developed for each sprint or each feature.

Use cases, stories and scenarios are all essentially the same thing. They are a description of a the use of the system for a particular purpose. They describe a common way the system will be used from start to end, providing the user's motivation, how they approach the system, what they provide and ask of the system, and how the system responds. They typically describe a sequence of actions by the user and the system.

They are several properties that make a good use case or story. A good use case needs to provide a detailed description of the task including the motivation. This is best done by being as specific as possible rather than stating what the system does in more general terms. For example, they should provide names for the user or users involved to make it personal; they should provide underlying details that describe the motivation or why the user wants to do a particular task. They should describe how the user approaches the system. They should detail the input and output provided. They might include details about what happens if the user makes a mistake and the system needs to respond with an error message. They should detail exactly what the system does in response to the user's actions, again using specific sample data for both input and output.

One way of thinking about such a scenario is to think of it as a short story or novel. Think about any novels you have read recently. They try to provide the reader with a feel for the situation to given them a better understanding of what is happening. They do this by providing details such as the what food is being eaten or what furniture is in the room. Most of these details are irrelevant to the plot but they are often necessary to understand the characters and the provide the appropriate ambiance. Similarly, a scenario needs to provide extra details to ensure the reader is on the same page as the writer in terms of what they are trying to accomplish, how they approach the system, and how the interpret the system's responses.

For the initial system and when developing a complete set of requirements, using an agile approach means developing a set of stories and scenarios covering the principle tasks the system is to perform from the point of view of multiple stakeholders. They should provide the developers with a sense of what they users expect the system to do, how the application will be used, and exactly what problems are being solved.

**[SPR: should we add a sample scenario here or defer to the SHORE section?”]**

Stories and scenarios are also a good basis for getting user feedback. Not only can users create scenarios, but developers can create scenarios for how they think users would want to use the system. These can be shown to the stakeholders. Since they are relatively self-contained and easy to understand, this provides a basis for stakeholders to tell the developers if they are on the right track and to modify the given scenarios to better match how they would actually use the system. Stories from the stakeholders can similarly be looked at by the developers who can then offer suggested changes or alternatives and see if they make sense from the stakeholder’s perspective.

While stories and scenarios are primarily written in natural language, UML includes a formalism to better describe the various interactions between people, various system components, and outside systems. This is the UML Use-Case diagram. Because it often talks about system components, such diagrams are more appropriate for specifications which can also be based on stories and scenarios. Use-case diagrams generally only show relationships. To provide more meaningful requirements, they need to be augmented with additional details as one would do with a scenario. **[SPR: Use consistent terminology once the 3 terms have been introduced. Possibly view a use case as something more formal?]**

### 3.3.3 Practical Requirements

In practice, a combination of list-based requirements and stories or scenarios work best. The list-based requirements provide the details needed by developers when they start doing detailed design and develop the actual code. These processes require a lot of specifics in order to build the correct system and meet the users requirements. On the other hand, stories provide developers with a better feel for what the system does and how it will be used. This is needed when building the user interface and determining priorities for the various requirements and how they are incorporated into the eventual design and system.

## 3.4 Requirements for SHORE

SHORE is designed to control a model railroad. As such, the primary stakeholders are people who own and use such railroads. Secondary stakeholders might include people who like to look at such railroads and companies that make controls for such railroads. For our purposes, we concentrated on the primary stakeholders.

Model railroads can be fun and challenging to operate. They can have lots of controls and offer multiple possibilities for what can be done with the layout. The challenges come from preventing harmful conditions such as collisions between trains and derailments while at the same time achieving some particular goal, for example having two trains running on the tracks simultaneously while preparing a third train to run when one of those is done. A complex layout might require multiple users to handle the controls simultaneously to account for the multiple trains.

Watching a model railroad in operation can also be fun. Railroad displays that have multiple trains operating simultaneously are featured in museums nationwide. (See https://www.trains.com/ctt/how-to/toy-train-layouts/toy-train-layouts-you-can-visit for an up to date list.) This includes science museums such as the Carnegie Science Center (and previously the Franklin Institute in Philadelphia), as well as a number of railroad museums.

SHORE is designed to let model railroaders provide the views of model railroad in action while still offering the challenges inherent to the hobby. It needs to ensure the safety of the rails. It needs to allow operation of a complex set up by a single user. It needs to provide a long-running display without requiring people continually updating everything.

### 3.4.1 List-Based Requirements

Based on this overview, we solicited a set of requirements as to what the target system should do from a user's perspective. These are shown in Figures 3.1 and 3.2.

These requirements are separated into sections. They start with general requirements that represent constraints on the target solution, for example what type of machine the system should run on, what type of train hardware is needed, what types of layouts it should handle, what aspects of the railroad are under the control of the system, and the fact that the system should allow both automatic and manual control. The latter aspect lets the user still have the challenge of controlling the trains manually when they so desire. The second section basically states that the

I. General Requirements
  A. Platform
    1. The system should run on a machine near the train table. It can be assumed to be running when the trains are active.
    2. The system should be able to run on mac, Linux or windows.
  B. Layout
    1. The system should accommodate a wide variety of layouts.
  C. Equipment
    1. Trains should be equipped with WiFi control using LocoFi controllers
  D. Operation
    1. The system should allow manual control of the layout and the trains as well as automatic control. These can be done simultaneously for different trains.
      a. Manual control of the trains can be via controllers for WiFi trains
      b. Automatic control should allow user to specify planned routes for each train
  E. User Interface
    1. The layout should be displayed as an abstract track.
    2. The state of switches and the state of signals should be included in the display
    3. The position of trains should be included in the display
    4. The user should be able to control switches and signals from the display.
    5. Train controllers akin to the LocoFi app should be (an optional) part of the interface.
    6. There should be an interface to define train routes for automatic control.

II. Layout Definition Requirements
  A. Layout Resource File
    1. The layout should be defined in a single user-editable resource file

III. Safety Requirements
  A. Switches
    1. If a train approaches a switch from one portion of the Y then the switch should be set to accommodate that.
    2. A switch should not be changed if a train is on it currently
  B. Track Blocks
    1. Only one train at a time should be in a track block
  C. Signals
    1. A signal can be associated with an upcoming track block. If it is, then it is red if the track block is in use or reserved for a train other than the one approaching it.
    2. A train should not pass a red signal.
  D. Trains
    1. It should be possible to set the maximum speed for a train for each portion of the track.
    2. It should be possible to indicate level crossings where the train should blow a whistle before crossing,
  E. Other
    1. Track (X) crossings should ensure that no train is in the block that is being crossed.

IV. Train Control Requirements
  A. Automatic versus manual control
    1. The user can manually control a train either through the (optional) user interface provided by the system or using the LocoFi train controller.
    2. Safety concerns should override manual control whenever possible.
  B. It should be possible to have consists (trains with multiple engines).

Figure 3.1: Requirements for SHORE – part 1.

system should be easy to adapt to a new layout using a user-editable resource file.

The Safety Requirements section defines the various problematic conditions that the system should check for and prevent. Switches need to be set correctly or trains are likely to derail. Track blocks are a common mechanism based on actual railroads and the given constraint (III.B.1) is the typical solution to preventing collisions. The section also details how signals should work, that train speed control is needed to prevent derailments at certain points in the layout, and that track crossings need to be handled.

The third section restates the need for manual versus automatic control and notes that the train control in the system should be similar to the existing controllers.

V. Display Requirements
- A. Contents
  1. The display should include the abstract layout that is used in standard descriptions such as track layout examples.
- B. Abstract layout
  1. The abstract layout can be split into multiple diagrams if the layout is complex.
  2. The display should show the position of trains as best it can.
  3. The user should be able to control the train layout from the display.
- C. Planning interface
  1. The user should be able to create and edit planned routes
  2. Routes can be saved and recalled
  3. The user should be able to assign a new or saved route to a train and have it execute that route.
  4. Routes can be aborted by resuming manual control.
  5. Routes can be paused and resumed.
  6. Routes can include timed stops (stations for example).
  7. Routes can be infinite (back and forth indefinitely).
  8. It should be possible to preview a route.
- D. Engine interface
  1. The engine interface should look and feel like the LocoFi application for normal control
- E. Mobile interface
  1. A mobile interface (phone or tablet) is optional

VI. Video Requirements
- A. Optional Video Input
  1. It should be possible to attach a video camera that has a complete view of the train table to provide additional information on train positions.
- B. Using video
  1. Video input will be used to provide precise information about train locations (start and end).
  2. Video input will be used to identify trains

VII. Hardware Requirements
- A. Base Machine
  1. The base machine should have WiFi and be on the same WiFi network (and switch) as the train layout and trains.
  2. The software should run on mac, windows, and Linux.
- B. Train table
  1. There should be WiFi controllers for switches, sensors, and signals.
- C. Engines
  1. Engines should use the LocoFi hardware.
  2. It should be possible to adapt the system to other hardware as it becomes available or when WiFi control becomes standardized.
  3. A WiFi to DCC controller can be defined if needed

Figure 3.2: Requirements for SHORE – part 2.

It also notes that the system should support train consists, a train with multiple engines.

The fourth section describes what is needed from the user interface. The emphasis here is on using existing display formalisms for model railroads and letting the the display be used to control the layout. The section also details what the potential users want to be able to do in terms of automatically controlling trains.

The fifth section describes an optional interface that the system should support. This involves a video camera focused on the layout that can provide additional information as to train position and behavior. The final section details the hardware requirements for the system. This is a elucidation of the general requirements on hardware.

Daniel has some experience with model trains, having been a big fan of Brios early on, and then having played with his grandfather's train set. Now he has done some supervised control of the train set using the existing LocoFi controllers and his smartphone. He wants to do something more sophisticated, running two trains at once, and wants to do it on his own.

Daniel starts up SHORE and manually puts together two freight trains in the yard area, one led by the BNSF locomotive and other by the SantaFe locomotive. He goes into the planning area of SHORE and creates a route that goes around the outside loop two times, over the reversing track, around the inside loop three times, and then back to the yard. He saves the route as "Long Haul", brings the SantaFe train to the edge of the yard, and then instructs SHORE to apply the route to that train.

As the train departs the yard, he uses the controls in SHORE to bring the BNSF train to the edge of the yard and then onto the outside loop. He traverses the outside loop with this train, noting that it periodically stops when it gets too close to the SantaFe train ahead of it. He continues until the SantaFe train has shifted to the inner loop, and then does one more turn around the outer loop, stopping to unload some freight. Finally, as the SantaFe train heads back to the yard, he uses SHORE to manually set switches to reverse to the inner loop and head back to the yard Finally he manually moves both trains to sidings in the yard using SHORE to control the engines and the switches and notes he has successfully run two trains at once.

Figure 3.3: Requirements for SHORE – Scenario.

### 3.4.2 Scenarios

The listed requirements provide the details needed to understand what the users expect out of the proposed system. This should provide a basis for defining what the system should do, its specifications, and then designing the appropriate system. However, they do not give a good sense of how the system would be used or what users expect when using the system. Such an understanding will help the developers design the correct system, better understand the various requirements, better understand what might be missing from the requirements, and better develop specifications. Combining the listed requirements with one or more scenarios can provide this additional information.

An example scenario is shown in Figure 3.3. This scenario shows a typical use of SHORE, illustrating both manual and automatic control. It is relatively specific, offering names, background, and motivation. It describes how the system would be used, but does not go into specific detail on exactly what the inputs and outputs of the system are.

We note that, as expected, the two representations for requirements compliment each other. The listed requirements provide specific details about safety and what users expect out of the system. The scenario shows how users might want to run the system and what they expect.

## 3.5 Non-Functional Requirements

So far we have talked about functional requirements, requirements that describe what the system should do and what users expect out of the system. Requirements analysis is also the time to consider what other constraints should be imposed on the system to ensure that the end result is a safe, reliable, and usable system. These constraints can involve security, privacy, fairness and other legal issues; they can involve reliability and performance as well as anticipated loads; and they can involve time and cost limitations on the development.

### 3.5.1 Security, Privacy and Fairness

Security has become a major issue over the past twenty years as any number of major companies have experienced security breaches, losing their and customer's private data to hackers. While it is almost impossible to create a perfectly secure system, it is possible to create a system where any hacker will have to expend more effort to access the data than the data is worth. Doing this, however, requires understanding the security and privacy concerns of the system in advance and taking them into account throughout the development process. Security is not something that can be added on, it is something that should be built into the system from day one.

Requirements should identify what private data the system absolutely needs to operate and should identify how securely that data has to be kept. It should note whether this data should be encrypted or not. It should note whether the data is needed long-term or only temporarily. It should also note any legal constraints related to the data.

Legal issues involving data are becoming more complex as countries and states enact privacy laws. Generally, any data that involves credit cards (CARD), health data (HIPAA), data about children (COPPA), or data about education (FERPA) is subject to legal restrictions. Any such data required or used by the system should be noted and the legal issues understood and made part of the requirements. European and state laws put other requirements on any personal data stored by the system, notably that it has to be available to the user and that the user can request that it be deleted. Any such requirements should also be noted.

If the requirements note different levels of users, for example, administrator versus normal user access, this should be noted and the requirements should indicate what safeguards should be in effect to ensure that the access levels are kept separate or that administrator or super users can not inadvertently release or compromise the system. For example, administrators might be limited to logging into the machine from within a particular network.

Privacy issues involve how a user's private data is used by the application. Do you plan to collect user history and use it for advertising or recommendations? Do you want to collect information about users to sell as part of funding the application? In these cases your application becomes one of the stakeholders and you need to include in the requirements what information needs to be recorded. This also affects security since the information then might need to be kept secure.

Another issue that is coming to the forefront with artificial intelligence and large-language models is fairness. There have been several instances where systems based on these technologies have been shown to discriminate against certain groups of individuals. In general, it is good to note, in the requirements phase of development, any concerns that might arise with respect to fairness so that these can be taken into account while specifying, designing and developing the system.

### 3.5.2 Universal Access

Another issue that should be considered as part of requirements is universal access. This has two parts. The first involves accommodating users with various disabilities, for example low vision, hearing loss, or wrist issues. If your application is consumer facing, it might be subject to the ADA laws which can constrain the user interface and hence the overall system specification and design. Requirements should note any places where such issues might be a problem to ensure that they are taken into account during further development.

There are standard techniques for ensuring these aspects of universal access, especially for web and mobile based applications. For example, W3C has developed an extensive set of guidelines and recommendations. While these are primarily concerned with the user interface design and implementation, noting what needs to be considered during requirements is still very helpful for later development.

The second part of universal access involves internationalization. If you anticipate that the application will be used in more than one language or in more than one country, than this should be noted as part of the requirements. It is much more difficult to add internationalization to an application after it has been developed than to develop it initially with internationalization in mind.

### 3.5.3 Performance

Understanding the expected performance and load requirements for a system can be crucial to developing the proper design. The design of a system that needs to handle a large number of users simultaneously can be quite different from the same system that only needs to handle a single or small number of users. Hence, understanding what is expected in terms of load early in the development process can lead to a better initial system and avoid the need to completely rewrite the system in the future. For example, SHORE is designed for a single layout and a relatively small number of trains. A system that had to handle hundreds or thousands of simultaneous trains might be designed quite differently.

Performance from the user's perspective is also important to consider during system development. Studies by Google have shown that interactive delays of as little as 100 milliseconds can be annoying to users and put them off the system. Part of understanding what users want out of the system is understanding what type of performance they expect. Such expectations should be included as part of the requirements.

Another related issue that can arise and should be noted as part of the requirements is reliability. How important is it to the stakeholders that the system is always available. For example, if the system is business critical and downtime means a significant loss of income, then the requirements should reflect that fact so that the system can be designed with enough redundancy or safeguards to minimize downtime.

Similarly, if any part of the system can involve problems in the real world, for example car crashes for a self-driving car, than these should be noted so that development can focus on them. Way back, we were developing a video game for an IMLAC computer. After having the display card burn out twice and needing to be repaired, we determined that, although the screen could be addressed from 0 to 08000, attempting to draw on the screen below 01400 or above 06400 would break the display card. Our requirements then said to ensure this never happened and our design and code added repeated checks for this. Similarly, when developing code to drive a pinball machine, we knew that attempting to trigger more than one solenoid or more than 8 lamps at one time would blow a fuse. Not blowing any fuses was part of the requirements and our design and code took this into account. More specifically, SHORE needs to avoid trains derailing which can cause them to fall off the table, or trains crashing which can damage the trains. These constraints are noted as part of the requirements so that they are fully considered during later development as a necessity and a focus of the design.

Finally, the overall development plan can impose constraints on the what the system can be expected to do. Funding for the system might be limited, which can limit what users should expect from the system. Similarly, the system might need to be available by a certain time. This also can limit how extensive the system can be and what users can expect of the system. Either of these should be noted as part of the requirements.

## 3.6 Obtaining Requirements

While the concept of requirements is straightforward, creating an accurate and complete requirements document can be difficult. One issue is determining what complete means and when the requirements are actually complete. A more central issue is that the developer needs to get information from potential users.

There are several ways that developers can obtain the information needed to create the requirements documents. The primary methods involve talking with the potential users, either directly or indirectly. Another approach is to observe users using their current approaches to understand what are the bottlenecks or difficult portions of their work and how it could be simplified through a new approach. Another approach involves building one or more simple prototypes to show potential users to get appropriate feedback. All of these need to be done multiple times. The first attempt will yield an initial view of requirements. These should be used to solicit feedback from the users to refine the requirements to ensure they actually meet the users needs and that the developer actually understood the users.

### 3.6.1 Talking to Users

Talking to users can be difficult. The primary problem is that users may not know what they want or why they think they want it. Users may not understand what potential software can do, either assuming some things would be too difficult for the software and hence should not be done, or assuming other things are simple for it and should be done even when those things are quite difficult from a software perspective and contribute little to the solution. It is the job of the developer to provide guidance to the user where needed so that both are on the same page.

A second problem is that users will often use a different argot, using common words that have specific meanings to their domain but can mean very different things to the developer. Developers need to understand precisely what users mean, asking questions to clarify anything that sounds odd or that they don't completely understand. Moreover, when developers are talking to users, they need to communicate in the user's terms, not in theirs.

A third problem is that users do not always know what they want even if they say they do. Users can be fickle since they do not understand what a potential system can and cannot do for them and since they can have difficulties imagining how the system will actually be used until the system exists.

One developer I know was writing video games for monkeys for a neurology lab. The original requirements said the monkey should be rewarded at the end of the game. The developer asked if it was ever the case that the reward should be given in the middle of the game. She asked this question not once, but multiple times during development. The answer always was no, only at the end of the game. Once the game was operational, the first modification that was asked for was to reward the monkeys in the middle of the game. Luckily, the developer had thought ahead and the change was relatively to make.

**[SPR: Paragraph on the use of LLMs and AI to analyze user input and produce a requirements document.]**

There are cases where developers want to skip the step of talking to users and develop requirements on their own. One case is if they are developing software for themselves or their team. Even here, it can be very helpful to talk to other developers to get a better sense of how to make the system more useful or appealing outside their little group. This often results in better, more robust software.

A second case is where the developer thinks they know better than the users as to what users want. There are a small number of people who have done this successfully, for example Steve Jobs at Apple. However, for most developers, this does not work. Note that even Steve Jobs had his company put a lot of money and effort into products that never succeeded, for example the Apple Newton.

### 3.6.2 Interviews

While it is possible and can be helpful to talk to potential users informally, and this can be helpful for getting initial ideas and an initial outline for a solution, a more formal approach with targeted questions and a specific agenda is generally more useful. This is the user interview.

Interviews are not easy. They require considerable preparation on the part of the developer. The developer has to understand the users and where the users are coming from, what their perspective is. They need to understand what information they need to obtain from the user. They need to convey enough information about their approach so that they can get this information.

Once developers have done their homework preparing for the interview, they need to decide on what type of interview is appropriate. This can be a structured interview where the developer asks a series of questions, one after another, to get the information they need. It can also be an unstructured interview, where the developer starts off asking questions but then follows the user's lead to determine additional questions and the direction of the interview.

A good interview will start with one or two general questions to get the interview on track and to ensure the interviewer and interviewee are on the same page. Then it will get into specific questions that are designed to elicit the detailed information needed to create the requirements. It will end with general or open questions, attempting to get the user to provide additional feedback, possibly giving the developer a view of the problem or solution they had not seen before. Once the participants are on the same page, it might be possible to solicit a scenario from the potential user as to what they might want the system to do.

If an interview is approached without having a good understanding of the target users and what to achieve, the results can be useless. A project group was aiming to develop an application to assist user interface developers. They approached four developers to try to understand what would work. The four developers came at the problem in very different ways, using different tools and strategies and with different backgrounds. The group got back four very different, and in many ways contradictory, answers. The result was not particularly helpful. A better approach would have been to first understand the problem they wanted to solve and then find developers who needed that problem addressed or to first attempt to get an understanding of how the developers worked rather than what the solution should be.

The developer should take notes during the interview, asking follow up questions when the answers might not be clear to them. After the interview they should write a summary of what they learned. This can the a scenario or just a succinct statement of what the developer learned. This should be provided to the interviewee in a thank you email to ensure it is correct and to solicit possible additional feedback. **[SPR: Sample set of interview questions for SHORE?]**

### 3.6.3 Questionaires

Interviews take considerable time and hence can only address a small set of potential users. An alternative is to create a questionnaire or survey that can be provided to a larger target audience to obtain a broader perspective on the problem and possible solutions.

Creating a survey requires more thought and effort than preparing for an interview. The survey needs to elicit the proper information without biasing the responses toward a particular solution. The developer is not able to clarify the questions so the user understands them correctly; not is the developer able to solicit further information based on a response.

In developing a questionnaire or survey, one should first decide who the target audience is to be. This can be any subset of the stakeholders. Different sets of stakeholders might require different surveys. The developer next has to determine what information they require from that particular set of users. To do this they need to have an initial understanding of the problem and of potential solutions. They then have to come up with a set of questions that provide that information, again without introducing bias. Finally, they have to determine how to interpret the results.

A typical survey will be a combination of rating questions and short answer questions. Rating questions using use a Likert scale, offering the user a range from 1 to 5 or 1 to 7 where 1 represents one extreme and 5 or 7 represents the other. Note that 5 and 7 are odd. It is possible to use Likert scales with even number, but users tend not to like them since they require a commitment on questions where the user has no real opinion.

The survey should start by getting a little information about the user, determining what group of stakeholders they belong to and what perspective they are approaching the problem with. Next it should ask the questions that are needed to gather the information needed for creating the requirements document. Finally they should end with open questions, asking the user to provide any additional input they might have on the problem or possible solutions. Note that not all users are going to provide such information.

There are several problems with surveys. Often, after getting the results of a survey, the developers will realize that they are missing key information. Either they forgot to ask a particular set of questions or the questions they did ask were not interpreted correctly by the users or the results of the survey were ambiguous or contradictory. Another problem involves how to interpret the resultant information. A survey might gather a lot of information about the problem, but might not provide enough information about potential solutions. It might also provide information about one solution, showing it has problems, but no information about possible better solutions.

A good approach to avoid these problems is to do a test run of the survey before making it generally available. Provide the survey to a small group of sample users, then see if the results are meaningful and useful for the project. Modify the survey to get better results and repeat this process until one is confident the results will help define the requirements. **[SPR: Sample survey for SHORE?]**

### 3.6.4 Prototyping

At times developers will have one or more concrete ideas about potential solutions to a known problem or have an idea about a potential system that they think users might like. These ideas are often difficult to convey when talking to users or in the introductory material for a questionnaire or survey. While some of this can be conveyed by offering the user scenarios for them to comment on, an alternative approach is to create a prototype that can be the basis for user input and feedback.

A prototype at the requirement stage is a dummy version of the system that contains the minimum needed to convey to the user a sense of what the resultant system might be able to do. It is a means for soliciting feedback, not an implementation of the system.

In this sense, the prototype does not have to even involve any code or functionality. A paper prototype, with a simple user interface showing how the system would work, can be sufficient and is relatively inexpensive to develop. Another alternative is to create a web page with limited functionality, or to do the same using a design tool such as Figma. The page should illustrate what the system would do but need not have any real functionality behind it. If these wont suffice, one can build a much simplified version of the system, with very limited functionality.

In any case, a prototype at this stage should be throw-away. You want to use the prototype to solicit feedback from the user. You do not want to be wedded to or committed to a particular prototype since this would bias the outcome and is likely to result in the wrong solution. You want to feel free to create multiple prototypes based on user feedback, trying to get it right. Coded prototypes are typically created in haste and the code quality is not sufficient for a large, long-lived system. Moreover the code structure and design is often not what you will eventually want to use once you understand all the requirements of the system.

If you are not willing to throw away your prototype, then you should not be doing prototyping. Instead, you should specify the requirements for a minimal but extensible system, build that system first, and then use user feedback on that system to extend it. Creating a minimal system involves prioritizing the various requirements.

Another way of effecting prototyping is to add one or more users to the development team to provide input and feedback throughout the development process. These users would provide feedback on the requirement and would work with the system as it is developed. This concept, developed in the Scandinavian countries and then incorporated into agile development, provides ensures that the system keeps in touch with user needs.

## 3.7 Prioritizing Requirements

A large, complex piece of software is not written all at once. Instead it is written incrementally. This means that developers will want to build a minimal system, get it working, and then extend it by adding features and capabilities.

Requirements analysis should focus on defining all the eventual capabilities of the target system. This lets designers take possible future extensions to the system into account during the design and implementation phases of development and to add the various features more easily and without compromising the existing system.

However, this means that the requirements will consider many more capabilities than are actually going to be implemented, especially in the initial system. To accommodate this, one needs to prioritize the requirements based on user input so that developers know what features to incorporate in the initial system and what features can be considered later on.

The easiest way of prioritizing requirements is to note, for each requirement, if it is *required*, i.e. needs to be part of the initial system; *optional*, i.e. users would like to see this but it is not essential immediately; and *long-term*, i.e. something that would be nice to have. At this point one does not need to get into finer detail.

Note that requirements is an ongoing process and the priorities are going to change, especially as users see what the system can do and get a better understanding of how they might want to use it.

**[SPR: Figure listing the priorities of the various requirements for SHORE. Just list the numbers for each tier. Alternatively add (R), (O), and (L)**

**to the requirements document and explain that here.**]

Prioritized requirements are used to later on during the design and coding phases to define two initial systems. The first is a minimal core system. This is the heart of the implementation and what is needed to support all the other capabilities of the system. The second is a minimal viable product (MVP), which is the minimal system that user's feel they would be able to use. We will get to these later on.

## 3.8 Summary

Write a summary.

## 3.9 Further Reading

Texts and tools for requirements analysis.

## 3.10 Exercises

Draw up requirements for a simple program (bouncing balls?)

Draw up requirements for your project

What would be your priorities for SHORE?

Develop a questionnaire for eliciting user input for SHORE.

# 4. Specifications

Requirements look at the problem and define a potential solution from the user's perspective; specifications extend this definition from the developers perspective, attempting to create a well-defined solution that can be used as a basis for design and development.

Specifications thus augment the requirements by defining the inputs, outputs, and commands that the eventual solution must handle. They provide a basis for creating an appropriate user interface for the solution. They note the risks involved in building the solution that need to be addressed during development. They specify constraints on the solution, both functional and non-functional. They provide a basis for determining if the solution will work and for understanding what work will need to be done.

## 4.1 Specifications versus Requirements

Requirements define how users see the system and what they expect of it. Specifications define what the application will do. They define this from the developers point of view.

Specifications are closely tied to requirements. Requirements form the basis for the specifications. Specifications are typically derived from the requirements by taking what the user expects out of the solution and mapping that to inputs, outputs and commands. This involves elaborating on the requirements by adding additional information. Specifications provide a clearer definition of the system which can be shown to the users to obtain a clearer and better defined set of requirements.

Thus there is a feedback cycle where requirements are used to derive the specifications, specifications are used to better define the requirements, the refined requirements are used to create better specifications, and so on. This process, *requirements* *specifications*, should repeat until the what the system will do is clearly understood, users have agreed that the proposed solution will meet their needs and are enthusiastic about using it, and developers are comfortable proceeding with a design for the system.

### 4.1.1 Specification Contents

Specifications define what the application will do. While they can mention other systems the application will work with as well as abstract components and modules of the system, they are more likely to talk about the inputs, outputs, and commands of the proposed application.

Detailing the inputs defines what information is needed by the application. This can be outside information, for example business information or, in the case of SHORE, a definition of the layout. It includes a description of how the information is used, where the information comes from, and where the information goes. Since much of the information comes from the user, the specifications need to describe how the user could interact with the system, describing potential user interfaces. They also need to describe how the system will interact with external systems.

Inputs and outputs by themselves do not describe everything about the system. Specifications also need to define the actions or commands that the system will perform. These specify what the user can ask of the system and thus form the basis for the user interface. They also specify what other systems can request of the system where this is appropriate.

Commands are described by defining their inputs and outputs and the mappings from the inputs to the outputs based on the current state of the system. A complete set of such commands describes the behavior of the system. The set of commands should cover the requirements. Any action described in a scenario or listed in the requirements should be associated with a command.

Commands can be invoked in many ways, from command line requests, to buttons or menus in the interface, to direct manipulation. How each command is done is the purview of user interface design. While it is not appropriate to do a detailed user interface design as part of specifications, the specifications should provide a basis for the eventual design, possibly by including one or more alternatives that offer suggestions to the user interface designer.

Specifications also need to define the underlying data that the system will operate on. Most systems to not operate in isolation, but instead collaborate with other systems and are built on top of existing or developed data sets. The specifications need to detail these data sets. They need to describe how the data will be obtained and what legal or privacy restrictions are included in the data. If the new system is designed as a piece of an existing system, the specifications need to detail the interface between the two.

For example, consider an application that provides recommendations to the user based on what others have done in similar situations, for example recommending a book based on their previous purchases by looking at the history of what others who had similar purchases have looked at. Such an application needs to have a data set that includes everyone's purchases. The specifications should detail this. Moreover, the specifications should contain a plan for bootstrapping this process so that recommendations can be offered even when the system is not widely used.

Specifications should also cover non-functional aspects of the requirements, providing details on these requirements that are particular to the implementation. For example, they might note that a database needs to be kept encrypted to provide the necessary privacy or security guarantees specified in the requirements.

### 4.1.2 Twitter Data Viewer

Consider a simple example application. I collected geolocated Twitter data for about 12 years, starting for a class on introductory data science, and just continuing. The result is a dataset of about 3 billion tweets complete with contents, author, longitude and latitude, state, zip code, and congressional district. In addition, I created an inverted index of the words so one can easily find all the tweets containing a given word. For the class, we wanted to create a simple interface that would let the students effectively ask questions such as show the number of tweets containing the word ”flu” over the last year by state.

The inputs to the program are the raw twitter data and the user queries. We had created a program that just connected to Twitter, issued a streaming query for geolocated tweets within rectangles defining the United States, and stored the result as CSV data in files where each file held a maximum of one million tweets. Looking through the raw data, we noted that there were multiple types of geolocation included and that some tweets did not have geolocation. We also knew that our rectangles include parts of Canada and Mexico. We also noted that there was some duplication in the reported tweets.

The program we wanted to create was a front end that let users, primarily data science students, issue queries against this database without knowing a database query language, view the results in terms of a color-coded map with a time slider, and then generate the raw data for their query in a CSV file they could analyze further using Excel. The program also had to clean up and load the raw tweets. After getting user requirements, we developed an initial set of commands:

- *Load.* This command loads the latest twitter data into the system. It cleans up the data, removing duplicates, normalizes the geolocation information, ensures tweets are from the United States, and adds information about zip codes, city, state, and congressional district.
- *Search.* This command takes as input a time range and a set of keywords. Its output is a set of tweets that becomes the users working set.
- *Refine Search.* This command takes as input the user's current working set of tweets, a time range, and additional keywords. It restricts the current working set of tweets by the range and keywords.
- *Sample.* This command takes as input a count and generates a random sample of that size from the current working set.
- *Display.* This command inputs the type of region (state, district, or zip code), a time step, and the user's working set of tweets. It results in a map showing showing the number of tweets within the time step for each of the regions.
- *Export.* This command generates a CSV version of the current working set.

This set of commands, along with other requirements such as that user's wanted the application available as a web page, was sufficient to design and develop the system. However, we did not include requirements or specifications on performance, and the resultant system, because the database had insufficient memory and relatively slow disks, was only usable if one was very patient since an initial query could take an hour or more to run.

### 4.1.3 Specifications and Design

We should emphasize that specifications describe what the system does. They do not describe how the system works. They do not identify specific technologies that should be used unless these are mandated by outside requirements. For example, for the Twitter application, the specifications did not specify the type of database, how the working set would be stored, what web technologies would be used, or what back end technologies would be used to support the web front end.

Specifications also do not determine how or where tasks should be done. For the Twitter application, they did not specify what parts of the process should be run in the browser, in the cloud, on a server, etc. They do not specify the algorithms or processing to be used, for example how keyword search was to be done.

Specifications also do not provide detailed user interface designs. Our original specifications for the Twitter application include sketches showing how the needed user input could be gathered and how the display might look, but these were very preliminary and are only vaguely related to the final design.

Specifications are there to define the solution. They are not a design or a description of how the application should be implemented. They tell you what the application will do, not how it will be done. When creating specifications, you do not want to over commit the implementation. Rather you want to understand everything the application should or might in future do so that the best design can eventually be created.

I've seen too many projects, especially student projects, where the developers choose the implementation framework or language or both early in the specification process. Often this decision was made prematurely and an alternative decisions would have yielded a more robust and appropriate system in the long term.

**[SPR: Should have text or a section here on data sources. Use examples of SPHERE-E failing because they had no plan for obtaining and maintaining the data that was key to the system. Also note that sites that depend on recommendations or on user feedback need a bootstrap plan. This should also be mentioned in risk analysis.]**

## 4.2 Representing Specifications

As specifications are closely tied to requirements, the representations of the two are also closely related. In particular, the easiest way of creating specifications is to extend the requirements document to include the additional information needed to define the solution from the developer's point of view. In addition, specifications should include information about the user interface, often requiring additional documentation.

I. General Specifications
  A. Platform
    1. The system should run on a machine near the train table. It can be assumed to be running when the trains are active.
    2. The system should be able to run on mac, linux or windows.
  B. Layout
    1. The system should accommodate a wide variety of layouts.
    2. Layouts should be defined in a user editable resource file
  C. Equipment
    1. Trains should be equipped with WiFi control using LocoFi controllers
    2. The layout should include a number of tower controllers, each managing a set of switches, sensors, and signals and connected via WiFi.
  D. Operation
    1. The system should initialize itself and not require any special positioning of trains to start.
    2. The system should allow manual control of the layout and the trains as well as automatic control. These can be done simultaneously for different trains.
    3. Manual control of the layout can be via existing switches on the layout or through the system
    4. Manual control of the trains can be via controllers for WiFi trains
      a. Automatic control should allow user to specify planned routes for each train
  E. User Interface
    1. The layout should be displayed as an abstract track.
    2. The state of switches and the state of signals should be included in the display
    3. The position of trains should be included in the display
    4. The user should be able to control switches and signals from the display.
    5. Train controllers akin to the LocoFi app should be (an optional) part of the interface.
    6. There should be an interface to define train routes for automatic control.

II. Layout Definition Requirements
  A. Layout Resource File
    1. The layout should be defined in a user-editable resource file
    2. This file can be either xml or json
    3. The file should provide the minimum information possible. Information that can be derived from the minimum need not be present.
  B. Resource File Contents
    1. The resource file should define the abstract layout as a set of points in an arbitrary coordinate system.
    2. Point locations can be approximate; proper alignment will be done by the system.
    3. The file should specify the location of each sensor, switch, and signal on the abstract layout.
    4. The file should specify at least one location specific to each track block.
    5. Each sensor, signal, and switch should have information as to what tower it is connected to and its identity within that tower.
  C. Additional Contents
    1. The resource file should be used to provide additional information about the layout in addition to the track layout
    2. The file should include the names of each engine and information about that engine
    3. The file should define the information needed for planning automatic control
    4. The file should contain any other information related to the particular layout that the system needs.

Figure 4.1: Specifications for SHORE – part 1.

### 4.2.1 List-Based Specifications

The simplest representation of the specification is a hierarchical list that is an extension of the requirements document. Here one takes the requirements document and adds new items and subitems that include the additional information about the what the system should do from the developers perspective.

These specifications can be created by stepping through the requirements document and determining, for each item, if the developer actually understands what the system could do to accomplish the requirement, and, if not, adding the additional information.

In this way, the specifications document is just an extension of the requirements

III. Safety Requirements
   A. Switches
      1. If a train approaches a switch from one portion of the Y then the switch should be set to accommodate that.
      2. A switch should not be changed if a train is on it currently
   B. Track Blocks
      1. Only one train at a time should be in a track block
      2. A train approaching a track block should reserve that block
      3. A train can only enter a new track block if it has reserved that block
   C. Signals
      1. A signal can be associated with an upcoming track block. If it is, then it is red if the track block is in use or reserved for a train other than the one approaching it.
      2. A train should not pass a red signal.
   D. Trains
      1. It should be possible to set the maximum speed for a train for each portion of the track.
      2. It should be possible to indicate level crossings where the train should blow a whistle before crossing,
   E. Other
      1. Track (X) crossings should ensure that no train is in the block that is being crossed.

IV. Train Control Requirements
   A. Automatic versus manual control
      1. The user can manually control a train either through the (optional) user interface provided by the system or using the LocoFi train controller.
      2. Safety concerns should override manual control whenever possible.
   B. If other WiFi controls become common or WiFi standard for train control emerges, then the system should adapt.
   C. It should be possible to have consists (trains with multiple engines) if the underlying engine controls support this.

Figure 4.2: Specifications for SHORE – part 2.

document. It is helpful, especially when going back to the users to get feedback on the proposed specifications, to maintain the original requirements numbering. This can be done by do any augmentation carefully or by creating a new document and adding references for each item pointing to the section in the original requirements document.

The list-based specification for SHORE, derived from the requirements documents, is shown in Figures 4.1, 4.2, 4.3 and 4.4. Comparing the original requirements provided in the previous chapter with these specifications, one notes that the specifications contain significantly more information, information that is relevant to the implementation and will be useful when designing and implementing the software.

**[SPR: Include priorities in SHORE specifications. Start with (R),(O),(L) from requirements document. Add these to specs. Possibly augment with R+, R-, O+, O-.]**

For example, in the SHORE specification, section II about the layout file basically says that the layout should be defined in a user-editable resource file. This is not enough information for the designer to work with. The specifications need to go into more detail, describing the format of the file, XML or JSON for example, what it will contain, and how it can be extended to accommodate the various extended capabilities that SHORE might provide.

V. Display Requirements
  A. Contents
    1. The display should include the abstract layout
    2. The display should include a planning interface
    3. The display should include, at the user's discretion, controllers for each of the engines.
    4. The display should be responsive and handle different window sizes and shapes.
  B. Abstract layout
    1. The abstract layout can be split into multiple diagrams if the layout is complex.
    2. The display should label each switch.
    3. The display should show the current setting of each switch.
    4. The display can optionally label each track block.
    5. The display should show the connections between track blocks.
    6. The display should show the relative position of each signal and its current state.
    7. The display should show the position of trains as best it can.
  C. Planning interface
    1. The user should be able to create and edit planned routes
    2. Routes can be saved and recalled
    3. The user should be able to assign a new or saved route to a train and have it execute that route.
    4. Routes can be aborted by resuming manual control.
    5. Routes can be paused and resumed.
    6. Routes can include timed stops (stations for example).
    7. Routes can be infinite (back and forth indefinitely).
    8. It should be possible to preview a route.
  D. Engine interface
    1. The engine interface should look and feel like the LocoFi application for normal control
    2. Emergency-stop should be reserved for the safety system.
    3. If other WiFi-enabled trains are included, the interface should accommodate those.
    4. There should be an interface for setting up consists.
    5. The secondary (configuration, etc.) LocoFi interface could be provided.
  E. RESTful interface
    1. The system should provide a restful interface that would allow a web or mobile application to control the layout.

Figure 4.3: Specifications for SHORE – part 3.

**[SPR: Other interesting things: dependence on outside packages. non-functional specs]**

One also notes that the specifications specify what should be done. They do not define how it should be done or define the resultant system. They do not constitute a design for the software, but rather provide the constraints for that design.

### 4.2.2 User Interface Sketches

Inputs, outputs and commands are an essential part of the specifications. Most modern systems make use of a user interface to communicate these with the system. This interface could be a mobile application, a web page, a workstation application, or something else (for example with Internet-of-Things devices). While the exact nature of the interface is best left to design, it is helpful to have a rough idea of what an interface might look like as part of the specifications. The goal here is not to create a high-quality user interface, but rather to ensure that an interface that handles all the requirements is possible, to give the users a sense of what the system might look like, to give the system designers a sense of how commands will be used, and to give user interface designers a starting point for their designs.

VI. Planning Requirements
- A. Plans
  1. A planned route should consist of a sequence of blocks defining the route.
  2. Routes can be infinite or continuous as well as one-time.
  3. Where necessary (say on a trolley going back and forth over a single line), the route should include direction information.
  4. Routes can include stopping points with appropriate delays.
- B. Planning
  1. The user should not have to specify all the information for a route. Where such information can be inferred (intermediate blocks, directions), the inference should be done by the system.

VII. Video Requirements
- A. Optional Video Input
  1. It should be possible to attach a video camera that has a complete view of the train table to provide additional information on train positions.
  2. The camera should be in a fixed position
  3. The system cannot assume that the camera has a complete view of the layout since there might be tunnels or other occlusions.
- B. Initialization
  1. There can be an initialization phase for using a camera where the camera identifies all the tracks, possibly by having an engine run over the complete layout.
  2. The actual track image should be correlated with the abstract layout using the sensor positions and approximating in between positions.
  3. The user should be able to identify pseudo-sensors by identifying a position on the video image. These should then be treated as regular sensors.
  4. The camera should be shown images of the different engines so it might be possible for it to identify particular trains visually.
- C. Using video
  1. Video input will be used to provide precise information about train locations (start and end).
  2. Video input will be used to trigger pseudo sensors

VIII. Hardware Requirements
- A. Base Machine
  1. The base machine should have WiFi and be on the same WiFi network (and switch) as the train layout and trains.
  2. The software should run on mac, windows, and linux.
- B. Train table
  1. There should be WiFi controllers for switches, sensors, and signals.
  2. These should initially be the controllers that we have defined and built, but the system should be able to adapt to other controllers easily.
- C. Engines
  1. Engines should use the LocoFi hardware.
  2. It should be possible to adapt the system to other hardware as it becomes available or when WiFi control becomes standardized.
  3. A WiFi to DCC controller can be defined if needed

Figure 4.4: Specifications for SHORE – part 4.

The easiest way of doing this is by creating a simple sketch of what the user interface might look like. The sketch should show the various user interface components, such as menus, dialogues, and drawn objects. It can provide pop-outs to give more detail, for example, listing the buttons on a pull-down or pop-up menu, or showing a dialog that results from a given button press. It can include multiple sketches to show the various pages of the interface.

**[SPR: Include sketch of SHORE user interface and a brief discussion of it here.]**

User interface sketches do not have to be complete. They do not have to cover all aspects of the application in detail. Nor do they have to be consistent or look good. If there are parts of the application that are not yet well defined, features that might be added in the future, then leaving room for a latter design of that portion of the interface is sufficient.

For example, the initial implementation of SHORE does not need to include the automatic control facilities. Creating a user interface for specifying the automatic control could be difficult, but is definitely possible. As part of the initial specifications for SHORE, we just dedicate a portion of the display for this interface, and do not provide any details.

### 4.2.3 Specifications Scenarios

Just as with requirements, the organized list, while providing details about the system, does not provide a good overview of what the system does and how it will be used. For requirements, we addressed this by augmenting the list with one or more scenarios that illustrate typical uses of the system. The same can be done for specifications.

A specification use case is generally an extension of a requirements use case. However, in addition to describing the users actions and the system output based on those actions, it describes the internal behavior of the system and any additional data that the system makes use of. It can also describe a sample user interface for the interaction and how it is used.

**[SPR: Augment the scenario with system information, either in line or as an supplemental description of what goes on in the system. Include it here as a figure. Provide a little discussion of what was done.]**

## 4.3 Risk analysis

Developing anything new involves risk. There is a risk the system will not work as intended. There is a risk that user's will not like or want to use the system. There is a risk the system cannot be built as specified. There is a risk the system will be out of date before it exists. There is a risk that the system will be too slow, too clumsy to use, or just now meet the user's needs. Developers need to be aware of the risks inherent to their particular system so they can proceed accordingly.

### 4.3.1 Risk Analysis

These risks can be divided into outside risks, things the developers can do little about, and inside risks, things the developers can address directly during the remainder of the development process. The specifications should concentrate on the latter since understanding these can help mitigate them.

The first step in the process is to do a risk analysis of the proposed system. This basically involves asking what can go wrong. Identifying risks can be difficult. The risks might not be obvious without considering the system and its eventual uses. The risks might depend on the development team. The risks might depend on the actual users of the system. The risks might depend on the environment in which the system is used. The risks might depend on future applications of the system.

Developers should start by asking themselves what can go wrong that they might have control over. One can identify several risks in the proposed SHORE system. First, the sensors are not perfect. Changes in lighting conditions might give a false sensor signal and it is unclear what the sensor state is between train cars. Hence the sensors should not be assumed to be 100% reliable. Second, initialization of the system can be problematic. If a train is over a sensor when the system starts, this might not be detected. Thus it might not be safe to trigger a switch at that point. Moreover, the initial state of all the switches is not known; and what value the various signals should have has not been determined. A third risk involves trains moving backwards. The system cannot assume that trains always move in the same direction, but needs to take into account that they might back up occasionally. Any system design needs to take these risks into account.

Next developers should identify the difficult portions of the system. Things that look difficult to design or build are things that one is likely not to get right the first time and will need to redo, possibly multiple times. Understanding what these are in advance will let the system be designed so that such changes are possible with a minimal impact on the remainder of the system. The user interface usually falls into this category, since it can be very difficult to develop a high quality interface before the system exists and the actual interface is going to change as the system evolves, possibly in major ways.

Another approach to finding risks involves developers asking themselves what they do not understand that they will need to in order to build the system. This can vary with different developers, but there are general issues for any development team. For example, with SHORE, no one on the development team has expertise in computer vision. Although the vision components needed are relatively simple (detecting trains on the layout), this is still an area where the team does not feel confident they can develop the correct initial solution, and hence one that is identified as a risk.

For different teams, risks based on understanding will vary. This might be what is the correct user interface or even the correct approach to a user interface. It might be what are the needs of actual users. Hopefully, the team obtained information from the correct set of users, but this might not actually be the case. Moreover, the users might not quite understand what they need. For other teams it might be the algorithms that are to be used. For example, if one wants to use large language models, is it sufficient to understand prompt engineering, or will one need to have a deeper understanding of the approach and customize a particular LLM to meet their needs.

### 4.3.2 Including Risks in Specifications

The information about potential risks is going to be needed in order to design a system that can accommodate and mitigate those risks. It is going to be needed to understand the effect of changes to the specifications and new features that might be added to the system. It is going to be needed when bringing new developers on board and even in deciding what types of developers might be added to the team.

All this means that the results of the risk analysis should be included in the specifications. This can be done either by noting them as subitems in the appropriate section or, probably better, by adding a separate risk section to the specifications. Each of the risks mentioned should then refer back to the appropriate specification. Just like the remainder of the specifications, this section should be maintained and revised as the system evolves and the risks are better understood.

**[SPR: Figure with risk specification section?]**

## 4.4 Prototyping

One way of managing risk and obtaining a better understanding of the target software is to build a prototype of part of the system. As with requirement's prototypes discussed in Section 3.6.4, these should be relatively quick and dirty implementations, possibly using high-level tools, that concentrate on the particular aspect of the system that is seen as a risk or where the best approach is not known. They should not be viewed as initial versions of the system.

Prototypes should be the basis for making informed decisions regarding the specifications and what will eventually work within the system. They should not be part of the final system and should be designed and developed to be thrown away. If you are not willing to throw away the prototype, you should not be writing a prototype.

Instead, design the system in a way that it is relatively easy to experiment with the different alternatives and then develop different implementations of the portions of the system in question.

We have done prototyping successfully several times. A start up was attempting to build a front end to search around 1999 that used WordNet **[SPR: Ref?]** to augment ones search queries in order to get better results from the then primitive search engines. We did not understand how well this would work, nor what user interfaces might be appropriate. We developed a prototype based on a Java Applet that included the basic functionality and a very simple user interface. This helped us specify and then design the actual system which was a web application with a JavaScript-based front end and a Java back-end. None of the original code was used in the eventual system.

A second instance was with the Code Bubbles integrated development environment discussed in Section 2.2.1. Code Bubbles is mainly a new user interface for code development. We developed a simple, partial implementation of this interface for code editing and browsing using Microsoft Windows Presentation Foundation, high-level graphics package. This gave us an environment where we could concentrate on the basic functionality and making the user interface look good. It allowed easy experimentation with different possibilities. The resultant environment was one that could be shown to users to get an understanding of what worked and what did not. This implementation was too slow to be practical, difficult to extend, and only ran on Windows, not on the Linux platforms we were mainly interested in. We used the ideas from the prototype to develop what has become the current version of Code Bubbles, written mainly in Java. The original prototype was thrown away.

## 4.5 Maintaining Specifications

The specifications are going to change. They are going to change as users understand them. They are going to change as the system is built. They are going to change as the system and its environment evolve. They are going to change as the circumstances change.

### 4.5.1 Changing Specifications

The first set of changes involves checking if the specifications actually meet the user's needs. If the specifications document is done separately from the requirements rather than by extending it, this involves *traceability*, tracking which specifications satisfy which requirements. This is typically done by adding to the specifications references to which requirements they are designed to meet. This is done both to understand why a particular specification is present and to ensure that all the requirements are covered by one or more specification items.

Traceability can also extend beyond specifications. It can be helpful in the design and implementation of the system to note why a particular design choice was made when that choice is related to the original specifications. This should justify the choice and, hopefully, prevent later programmers from undoing it when it remains necessary.

If the specifications document is an extension of the requirements document, then checking if user's needs are still met can be done by showing the specifications to a set of users. Providing additional details of what the proposed system will do and how it would work will provide the users with a better understanding and will encourage them to think deeper as to whether the solution is actually appropriate and to develop alternative or additional suggestions on the system. These additional requirements can then lead to better specifications.

Note that this cycle of showing things to potential users, getting user feedback and suggestions, and then modifying the target system accordingly should be an on-going process. It is one reason why many development teams like to include a set of users, either directly on the team or ones who are anxious and willing to work with early products.

### 4.5.2 Prioritizing Specifications

Since large applications are not and cannot be built all at once, it is important to prioritize the specifications. This follows the prioritization of the requirements discussed in Section 3.7. Each of the specifications should be prioritized as either ”required”, ”optional”, or ”long-term”. Required items are those that would have to be in a system to make it at all usable. Optional items are those that would be helpful and would be needed to make the system competitive and attract other users. Long-term items are those that might be helpful and make a big difference in the future, but probably are not needed in the early versions of the system.

The idea of defining a minimal usable system is now well accepted, especially by start-up developers. This initial system is consider a minimal viable product or MVP. The idea is to get the new application in the hands of users as early as possible. This lets the developers get early user feedback, make any necessary changes earlier in the development process, determine what features are important to add next, and generally orient the overall development to meet the users' needs.

Having a MVP also helps developers. It provides a development goal that can be achieved early in the overall process. This will give developers confidence that the system can and will actually work. It will also provide a basis for developers to obtain their own feedback, for example, on what portions of the system seem to be too slow or too complex. It will also provide a way for the user interface designers to assess their initial user interfaces and to redesign these as necessary.

### 4.5.3 The Specifications Document

While it can be a good deal of work, maintaining an up-to-date specifications document throughout development has advantages, especially early in the development process.

The specifications document is the basis for the design. Design decisions, from the highest levels down to low level code details, need to take the constraints and concepts in the specifications in mind. There are often multiple ways of accomplishing a task, all of which will work. However, some of these are going to be better than others because they will directly address some aspect of the specifications or will make future development, again based on the specifications, simpler. Understanding and being able to go back and reference the specifications document can be helpful here.

The specifications document can also be a guideline for determining and prioritizing new features. The priorities listed in the document should then include ”done” indicating the specification is incorporated into the system. Rearranging priorities based on the various types of feedback should yield a new set of required items that are the candidates for the next set of features to be added to the system.

Finally, the specifications document can be helpful to provide developers with an overview of the system and why it is the way it is. This is especially true when new developers are brought on board during the development process. These newcomers are generally not aware of the history of the system or why things were designed and built the way they were. Understanding the specifications, in their current form, can help the new developers build up this understanding.

## 4.6 Summary

Creating a well defined specification is essential to building the actual system. Specifications are going to change based on user feedback and as the system and its environment evolves.

## 4.7 Further Reading

## 4.8 Exercises

Create specification for a simple project. Prototype? User Interface sketch?

Create specifications for the software you are developing.

# Design

# 5. Software Architectures

Requirements tell us what users need built. Specifications tell us what to build to meet the users' needs. The next step is to determine how to build the target system. This is design.

## 5.1 Software Design

Software design involves making a lot of decisions. Some of these are big decisions, for example what should be the overall architecture of your system. Some of these are small decisions, for example should we use as hash map or a tree map for this data structure. Most of the decisions do not matter that much. Anything can be made to work and anything can be redone if necessary. However, each of these decisions, if made correctly, can contribute to making the code and the overall system better.

Better code implies a better system. Better code means the system is easier to maintain. It can be extended and evolved as needed. Features can be added without rewriting large portions of the system. Better code means the code is simpler and easier to understand. This makes it easier for other programmers to work with the code and makes the code easier to debug. Better code is less subject to risk. It might alleviate risks or it might encapsulate them, allowing different alternative to be tried without affecting the rest of the system. Better code is easier to work on as a team. Any programmer on the team should feel comfortable working on any code in the system.

For each design decision, developers need to ask themselves what is the effect of their decision on the system. They need to understand the consequences and implications. They need to understand how the decision fits with the solution, both the current implementation and in the future. They need to consider whether the decision make the implementation easier or more difficult, both now and in the future. They need to consider the simplicity of the result in terms of the size of the code, the complexity of the code, and the complexity of interactions between the particular code and other parts of the system. They need to consider whether the decision makes the code easier or more difficult to understand. They need to consider whether the decision is consistent with similar decisions made in the rest of the system. They need to consider whether the decision might affect performance and if so, whether that performance is important or critical. They need to understand if the decision might raise security or privacy concerns.

A good designer is one who can make these decisions correctly most of the time. They do this based on experience. Experienced designers have read lots of designs and understand what others have done and what worked well in other cases. They have made a lot of design decisions before on other systems and have an understanding of what worked and what did not in similar cases. They remember their past decisions and their effects. When they use a new system, they are curious and think about how it might be designed. If the system has an odd behavior, they think about how the design might have created this behavior and if the design could have avoided the problem.

A good designer thinks in terms of design patterns. Patterns codify how things are done and offer standard ways of doing things that work. A design pattern basically defines a problem that requires a design decision and offers a solution to that problem. It includes information about when the solution should be used, that is, what in particular it is good for. It also includes information about when the solution should not be used, what consequences it might have that could be undesirable. Design patterns exist at all levels and need to be understood at all levels.

Design decisions are made at all levels. At the code level one can consider whether to use a for or a while for a loop, or can consider what data structure or algorithm to use. Decisions are made as to how to split the code into classes and methods. Decisions are made as to how to organize the code into packages or modules. Decisions are made as to how and when to use outside code. Decisions are made as to how to create common code that can be used by others. Decisions are made detailing the overall structure of the system. The high level decisions on the overall structure are the ones we are concerned with in this chapter. These define the *software architecture* of the system.

**[SPR: Possibly put in forward references to later chapters in the above paragraph.]**

## 5.2 What is Software Architecture

Wikipedia defines software architecture as:

> Software architecture refers to the fundamental structures of a software system and the discipline of creating such structures and systems. Each structure comprises software elements, relations among them, and properties of the elements and relations.**[SPR: cite]**

This definition emphasize that software architecture deals with the fundamental or high-level structure of a software system. It also notes that the architecture deals with more than the elements composing that structure in that also includes the relations between the elements.

Along similar lines, IEEE defines software architecture as:

> Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution. **[SPR: cite]**

This definition indicates that software architecture deals with the system organization and again emphasizes that architecture is more than just the various software components involved It includes their relationships. It extends the original definition by noting that architecture also deals with the environment surrounding the software and provides a set of principles for how to create and evolve the high-level software design.

Kruchten provides a more extensive definition:

> An architecture is the set of significant decisions about the organization of a software system, the selection of structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these elements into progressively larger subsystems, and the architectural style that guides that organization – these elements and their interfaces, their collaborations, and their composition. **[SPR: cite]**

This definition notes that software design is the process of making decisions and again emphasizes that the architecture is more than just compositional elements, noting that their collaboration is just as important. It also notes that architectures can be composed into progressively larger systems, hinting at the fact that most larger software systems do not follow a simple software architecture, but instead are compositions of multiple architectures.

In general, these definitions convey that software architecture is an overall view of how the system under development should work. It is a high-level definition that is needed when the system is complex. Moreover, the definitions highlight the fact that the architecture is expressed as components and their interactions.

Software architectures are typically represented in terms of boxes and arrows. The boxes represent the various system components, while the arrows represent communications between these components. These communications can be interpreted in various ways, such as direct calls, web accesses, messages, queued pipelines, etc. Moreover, the communications and hence the connections can be software components in their own right; they can involve significant processing as well as pure communication.

## 5.3 Architectural Patterns

Software design involves understanding patterns, when they are applicable, and when they are inappropriate. Patterns exist at all levels. In particular one can understand software architecture by understanding the most common architectural patterns.

There are a relatively small set of basic software architectures. The set isn't static – occasionally new architectures are developed and, at other times, older architectures might be rendered out of date. These various architectures can be represented as architectural design patterns.

High-level design patterns exist in many domains, including bridges, houses, electronics, etc. In these other domains it is not uncommon to find instances that combine multiple basic architectures. A complex bridge might be part arch and part truss. A house might be part farmhouse and part ranch. Similarly, a complex software system can be a combination of one or more architectural patterns or their variations.

A good designer will understand the different architectures. They will know what they are, how they might be implemented, when they are useful, what are their strengths, and what are their weaknesses. Below we describe some of the basic software architectures that are commonly used in modern software systems.

**[SPR: Reorder in a meaningful way? Ensure each section give the cases where the architecture is applicable and some indication of when it will have problems]**

## 5.3.1 Streaming Architectures

The pipe and filter architecture applies to systems where data is processed one step at a time. It is modeled on the UNIX shell pipe mechanism that allows the output of one program to be passed as input to the next. A pipe and filter system consists of filters which are essentially processing units, and pipes which are the connections between them. It is best used for a system that operates in phases.

The classic example of a pipe and filter architectures is a compiler as shown in FigureX. **[SPR: Add simple compiler architecture figure]** The first stage of a compiler reads the source file and generates a sequence of tokens. The second stage takes that sequence of tokens and builds a parse tree. The third stage performs semantic analysis on that parse tree and yields an annotated parse tree that includes symbol and type information as well as use-reference links. A fourth stage takes the annotate parse tree and generates an intermediate representation, usually a sequence of abstract instructions organized into basic blocks. A fifth stage performs optimization on the sequence of abstract instructions, yielding an optimized sequence. The sixth stage converts the abstract instructions into assembler code. A final stage converts the assembler code into a binary executable file.

Actual compilers are variations of this design. Sometimes two or more stages are combined; sometimes the stages yield complete outputs before the next stage begins while at other times the output is processed as it is generated; some stages are optional and omitted; some compilers have a separate symbol table data structure that is shared between stages while others embed the symbol information into the abstract syntax tree or intermediate representation. Sometime the pipes represent simple data streams as in the stream of tokens, while at other times the pipes are complex data structures that can include significant processing, for example abstract syntax trees with error checking. However, the overall structure of a compiler typically follows this architecture.

A more recent example of a pipeline architecture is our automatic program repair facility in Code Bubbles, ROSE. ROSE is designed to work in conjunction with a debugger in a programming environment. It is started at a breakpoint. The architecture of ROSE is shown in Figure 5.1. The first stage of ROSE is to get information from the developer about the symptoms that indicate a problem, for example an exception being thrown or a variable having the wrong value. The output of this phase is a description of the symptom. The second stage does fault localization based on this symptom. It uses an abstract-interpretation based flow analysis to do a backward slice from the stopping point to find a set of lines that

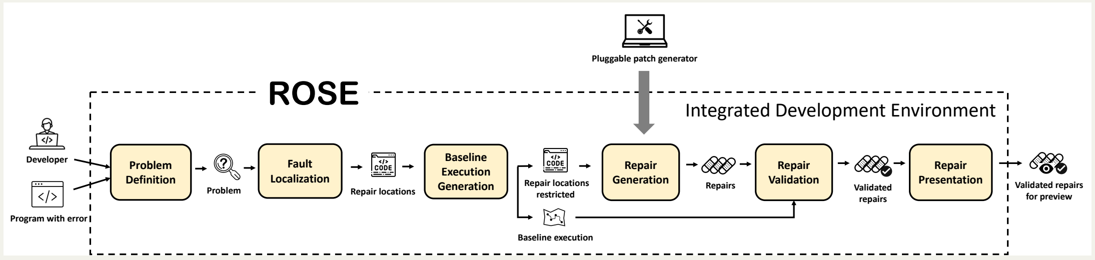


Figure 5.1: Architecture of ROSE

might have caused the symptom. The output of this stage is a set of potentially buggy lines as well as the problem description. The third stage generates a baseline execution trace using a live programming facility starting at some point on the current call stack. The trace needs to include as many of the potentially faulty locations as possible and duplicate the original symptom. The execution trace is used to remove potentially faulty locations that were not executed and hence cannot cause the problem. The result of this phase is the baseline execution, the restricted set of potentially faulty lines, and the original problem description.

The fourth stage uses a variety of tools including a pattern based analyzer that looks for common programming errors and an interface to ChatGPT to generate potential patches at the identified locations. The output of this stage is a set of potential patches, the base execution trace, and the original problem description. The fifth stage uses the live programming facility to generate a trace of the patched execution for each of the potential patches and then compares that execution with the base execution to see if the symptom disappeared and if the program seems to run correctly. The output of this stage is a set of patches that seem to work along with scores that describe the system's confidence that the patch is correct. The final stage presents these patches to the developer, lets them inspect the changes, and lets them make the patch and continue execution.

A generalization of this architecture allows data to be passed from one filter to multiple pipes and to have nodes that take multiple pipes as input. These generalized architectures are essentially data flow diagrams where the data is passed from one operator to another and the operators decide when they can process the inputs and generate output data.

There are several examples of such generalized streaming systems, often with graphical interfaces to let users connect the various components which can represent filters, data processing of various sorts, or outputs. Early work in 3-D modeling and visualization came from the Application Visualization System **[SPR: ref ieee paper]** where the processing nodes offered a variety of graphical options and effects. Another early work, still being used today, is the LabView system from National Instruments **[SPR: ref]**. This is a full-fledged data flow language where the processing elements can represent either program constructs or laboratory devices.

A more recent example along these lines are data mining systems where the input data can come from multiple sources, various operators can be applied to analyze and compress the data, and then different types of visualizations can be derived from the analyzed data. **[SPR: ref]** Another recent example are streaming databases where the nodes represent database operators that handle streamed data. **[SPR: ref]**

### 5.3.2 Layered Systems

While streaming architectures provide an obvious example with direct applications, the number of applications is not that large. A much more common architectural pattern is a layered system. Here the system is built in layers, with an inner core and then extensions that add functionality to that core. Many systems can be viewed this way.

A layered system can be viewed as an onion. An onion consists of an inner layer and then multiple layers surrounding that until one gets to the outside skin. In a layered system, the innermost layer represents the core functionality that is needed by everyone. A classic example is the way that networking is implemented in most systems. The innermost networking core is the physical layer that lets raw bit streams be transmitted over some medium. The second layer, data link, breaks data into frames. The third layer, network, provides a point-to-point link between nodes (UDP). The transport layer, built on top of this, manages the transmission of data between nodes, ensuring data arrives in the correct sequence and errors are corrected (TCP). The session layer then handles sessions between nodes, managing setup, authentication and termination. The presentation layer is responsible for translating data from network format to application formats (e.g. handling byte ordering). The final layer, the application layer, provides end-user protocols such as HTTP.

A typical example of a layered software system is the Java architecture shown in Figure 5.2. The inner core of Java consists of the Java HotSpot which supports the Java virtual machine both by interpretation and by compiling byte code into executable code. The next layer consists of the standard Java libraries (java.lang)

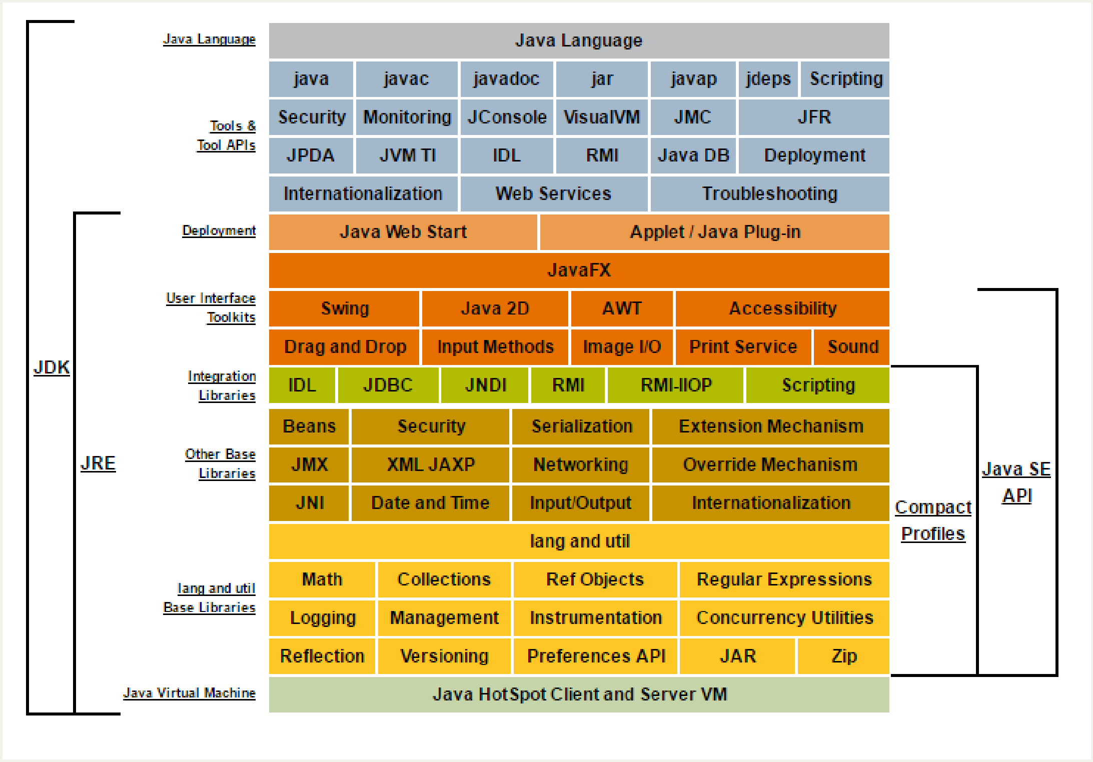


Figure 5.2: Java Architecture from Oracle

that are essential to executing Java programs. The subsequent layer contains the java.util libraries providing additional functional such as collections and various concurrency operations. Above this are the various libraries to provide extensions such as internationalization, date and time parsing, networking, and input-output. The next layer contains integration libraries, for example to access databases and remote procedure calls. Above this are the graphics layers, with AWT at the core and Swing and JavaFx built on top.

A variation of the onion approach is to view a layered system as a hub and spoke system. Here the innermost layers form the hub of the system. Extensions to this core are viewed as spokes in that they make use of the hub, but generally do not work directly with each other. Spokes are often viewed as optional extensions. Communications between the spokes, since they cannot rely on other spokes being present, are generally done through the hub. This simplifies the implementation in that the

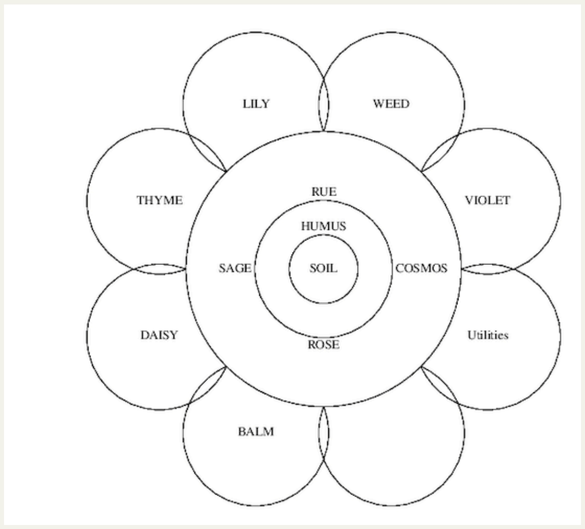


Figure 5.3: The architecture of Garden

spokes can be developed and changed independently.

We used a hub and spokes design in our Garden system as seen in Figure 5.3. Garden was a system for experimenting with visual languages. **[SPR: cite]** Programs were composed of executable objects where it was possible to define the executable semantics of each class of objects as well as their graphical representation. For example, one could define objects representing the states and transitions of a finite state automata, provide execution semantics based on finite automata, and then define a graphical representation which circles for states and arcs for transitions. The user could then create new finite automata using the graphical editor and have the result be executable.

The central core of GARDEN was and object-oriented database system, SOIL, that supported the set of objects with persistence (much like Smalltalk). Above this was the kernel of the Garden system, HUMUS, that defined the basic data types and their operations, the execution mechanisms, name spaces, dependencies and threads of control. It also defined a basic set of control flow objects for building programs. On top of HUMUS, was a set of 4 packages: SAGE and RUE provided basic graphics operations and callbacks, ROSE provided an interface to the operating system to support I/O, and COSMOS compiled object executions down to C code for faster execution.

Above this hub was a set of spokes that provided additional functionality and made Garden a usable system. These included LILY which provided a textual programming language; THYME which provided a type editor to define classes; DAISY and WEED which provided editors for a single object; BALM which provided Smalltalk-like browsing capabilities; VIOLET which provided an editor for the graphical representation of objects; and a utilities package provided a general text editor, a clock, and a doodling package. These did not depend on each other, but rather made use solely of the capabilities of the hub.

An extreme version of a hub and spokes architecture is a plug-in architecture. A plug-in system consists of a core and a mechanism for adding independent components that can make calls to core functionality. The plug-in components can be provided by the original developers or can be developed by outsiders. Plug-in architectures often provide facilities for dynamically loading and integrating the various plug ins, with the integration being done at run time rather than compile time. The core can define extension points which allow the plug-ins to augment the user interface or other capabilities provided by the core. The core also defines a set of callbacks whereby the plug-ins can be notified when things change and can take appropriate action.

A classic example of a plug-in architecture is the Eclipse integrated development environment. Eclipse includes a core that includes most of the basic functionality such as editing, compiling and debugging. The core offers the ability to define plug-in extensions and a wide variety of extensions have been developed, many of which are now considered central to Eclipse. For example, the plug-ins include interfaces for testing, version management, and collaboration. Moreover, Eclipse provides an active marketplace where developers can offer new plug-ins, either for free or for sale.

### 5.3.3 Repository Architectures

Another software architecture can be applied to systems where there is a shared common data structure that is central to the system. The system typically operates by making changes to this structure and then using the structure to drive various operations and the outputs. Typically, there is a trigger mechanism available to inform various components when something changes. This lets components that do input just change the central representation and lets components that do output or processing update automatically when a change is made.

A classic example of a repository architecture is a simple programming environment. Here the source code, either as text or as abstract syntax trees, forms the core

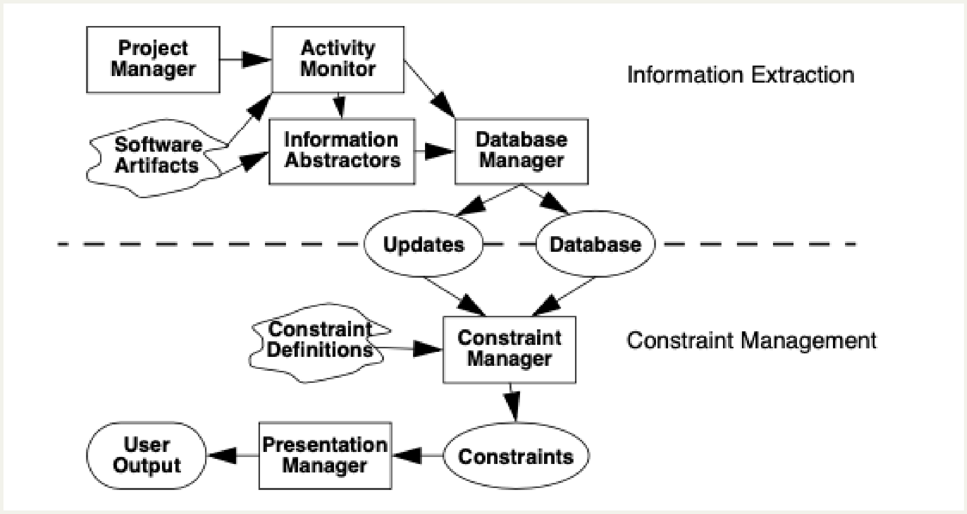


Figure 5.4: The architecture of CLIME

of the system. We used such an architecture in the early PECAN development system [SPR: ref] where the core representation was abstract syntax trees. The editors supported both direct structured edits to the abstract syntax trees and text edits which were parsed directly into changes to the trees. Structured edits could be done either on the source code or on flow-chart style views. Changes to the abstract syntax trees triggered callbacks which caused semantic analysis and corresponding error feedback in the editors. Changes in any of the views were automatically reflected in the other views since they were all based on the same common representation. A second common representation provided execution support. This contained the current run time stack and the values of variables. Execution was done by interpreting the abstract syntax trees and updating the run time representation. Different views, both specific for run time and source views, were updated based on callbacks when the execution representation updated. The execution representation also included the history of the values to support reverse execution.

The central repository does not have to be an in-memory data structure as was used in PECAN. It can be a database. We used a central SQL database in the CLIME system which attempted to inform programmers when the different representations of their software, for example code, documentation, and UML, were inconsistent. [SPR: citation] The architecture of CLIME is shown in Figure 5.4. The system was basically done in two parts. The first portion extracted information from various types of software artifacts and stored that information in a database. This portion would automatically scan the source code, documentation, and design documents as the corresponding files changed. It would map these into a common data structure representing the corresponding information and would store it in the database. A set of consistency constraints, defined as queries over this database provided the notion of consistency between the representations. The second part of the system was a constraint manager that used these definitions to derive the actual set of constraints based on the database. The constraints were automatically checked as the database was updated. Any inconsistencies that were detected were then passed to the display manager to be shown to the user.

Many modern web-based systems use this type of architecture. These systems typically use a database to store information about products, users, and program data. The front end uses a set of RESTful operations which are translated into updates and queries to this central database. The front end can either be a web browser or can be an application running on a mobile device. The back end web server, does the translation to database operations and also does further analysis, reports, and processing based on the data in the database, often updating as the database changes.

An extreme version of a repository architecture, and one that is becoming more common today, is one based on microservices. Microservices are loosely coupled small applications that talk to a central database rather than to each other. Microservices are smaller pieces of code that can often be reused between applications. They serve as an interface between the front end and a database and replace a comprehensive back end. With a microservice architecture each of the services handles a particular request, typically a URL with associated data. The service either translates the incoming data into a database update or, based on the incoming data does a database query and returns the result. The front end is responsible for creating the request and for formatting the returned output, typically using a package such as React or Vue.

Microservice architectures are flexible and relatively easy to scale. They are easily built by a team of programmers since each of the services, once well defined, can be created and maintained by a single programmer. They are a good choice for a web application that does not do a significant amount of back end processing, does not need to cache any intermediate results, and where the URLs alone can indicate what processing is required. They can involve extra work in the back end and duplicate code, for example each of the services will need to do its own authorization. However, microservices are at a level where they are relatively easy to reuse in new applications.

### 5.3.4 Interpreter-Based Architectures

Some systems are based on a kernel that is designed to be more general than the particular system in mind. Such a kernel lets the developer build a variety of different systems without having to reengineer the heart of the code. In these cases the kernel can be viewed as a programming language that is interpreted and the developed systems are viewed as being written in that programming language.

An important point here is that we are using the term programming language loosely. Some interpreter-based systems are built using a real programming language, essentially a domain-specific language which is oriented to a particular domain and for which the code is compiled or interpreted directly. However, for a majority of these systems, the ”programming language” is simply a library and programming is done by making calls to that library.

Domain specific languages have typically been used for programming scientific computing. The underlying libraries in this case use complex data types, typically a variety of matrix and vector representations, and there then provide a set of mathematical operations on these representations that are designed to be efficient. A domain specific language is used to hide the complex data representations and to let the system work easily on different architectures.

Library-based interpreters today generally use Python as the scripting language and provide a variety of calls that can be made from Python. The actual data structures are again somewhat hidden since the actual library code is written in a compiled language such as C or C++ and either the data is kept in the lower level form or is translated on each call. Since the actions implemented in the library can take considerable processing and time, the translation between languages is not a deterrent.

Such Python-based scripting is used for computer vision using the OpenCV library. OpenCV provides a set of data structures, most commonly a matrix representing an image where the actual structure is hidden from Python. It provides a large set of calls that can be used to do high-level processing of the image. Machine learning systems such as Apache spark provided a set of algorithms and data structures for large-scale data processing and machine learning to let one build AI applications. Again, the front end for such applications was written in Python. **[SPR: Diagram, sample opencv program?]**

Interpreter-based architectures are generally a good idea when one needs to build a variety of different high-level systems that make use of a common set of algorithms and data structures and where the algorithms and data structures can be complex and might require extensive optimization.

### 5.3.5 Client-Server Architectures

The availability and advantages of distributed computing have introduced a number of software architectural patterns. Some of these, for example, those based on a web server or microservices are often based on a central database and are best viewed as repository architectures. Others do not involve a central database, but instead involve one or more actors that communicate with each other.

When there is a principle component that serves as the controller for the other components, the result is a client-server architecture. A typical example of a client-server architecture, and one that for which client-server architectures have evolved, is a multiple-player computer game. Here there is a central server that manages the game and acts as an arbitrator, controller, and referee. The server runs on a centrally accessible machine and can be connected to using two-way sockets. The various players run a client program on their personal machine. The client program provides a graphical (now usually 3-D) interface and lets the players take actions, for example moving from room to room or shooting at something. The actions are sent to the server so it can keep track of the state of the game. The game state, taking into account these actions, including the location or actions of other relevant players are then sent back to the client to update its display accordingly. **[SPR: Diagram showing client-server architecture for a game]**

There are many variations of client-server architectures, generally based on what work gets done in the clients versus what gets done in the server. In some distributed games, almost all the work is done in the server and the client is only responsible for creating the graphics and handling user interaction. In other games, the client is relatively self-sufficient and much of the computation is done in the clients, with the server providing the necessary information from the other clients and ensuring that the clients remain synchronized. This requires carefully coding the client so that different clients will end up with the same result. Another variation involves how and when clients communicate with the server, with some games allowing clients to communicate at any time and others requiring them to take turns.

Client-server systems with separate client programs have become less common as web applications, and particularly web front ends based on JavaScript have become more powerful and flexible. Separate client programs have the ability to access the user's machine and resources such as files, cameras, networking, Bluetooth, and hardware acceleration. Modern browsers are getting access to these capabilities and it is becoming easier to write applications using a web front end rather that a using an application running on the user's machine. Today, when a client-server application is needed, it it typically built as a web application, with the server running as a web server and the communication from the clients being requests to the server. When asynchronous notifications from the server to the clients are needed, either the client makes continual requests or, more likely, web sockets are used.

Many applications written for mobile devices, phones and tablets, are essentially client-server applications, with a server running somewhere in the cloud and the front end running on the user's mobile device. These applications are generally built similar to web applications to allow a web-based front end as an alternative. They use HTML communication between the front end and the server. Systems like Dart and Flutter from Google **[SPR: ref]** support writing a common front end that will run in a browser, as a mobile application on Android or iOS devices, or as a native application on a workstation.

### 5.3.6 Message-Based Architectures

Client-server architectures assume there is a central server that controls the system. A more distributed approach is possible using a messaging. A message based architecture typically includes a message server that all the components communicate with, although it is possible to do messaging with only peer-to-peer communications. Components can register for the messages they are interested in. Components can then send messages to the message server which forwards them to all components who have expressed an interest. Messages can be simple notifications which do not require a response or can be commands or requests that do require a response.

The classic example of a message-based architecture is the FIELD programming environment **[SPR: citations]** whose architecture is shown in Figure 5.5. FIELD used a central message server and string-based messages. Various tools could register for messages of interest by providing string patterns for the messages they were interesting in seeing. FIELD provided wrappers for a variety of UNIX tools that translated outputs into messages and mapped messages into commands. These include tools for version management, tools for profiling, tools for debugging, and tools for cross referencing. In addition, FIELD provided its own tools that offered graphical interfaces for editing, browsing, and debugging.

Standard message services, more sophisticated than those provided by FIELD, were then developed and are still available. CORBA was one standard **[SPR: ref]**. Today,

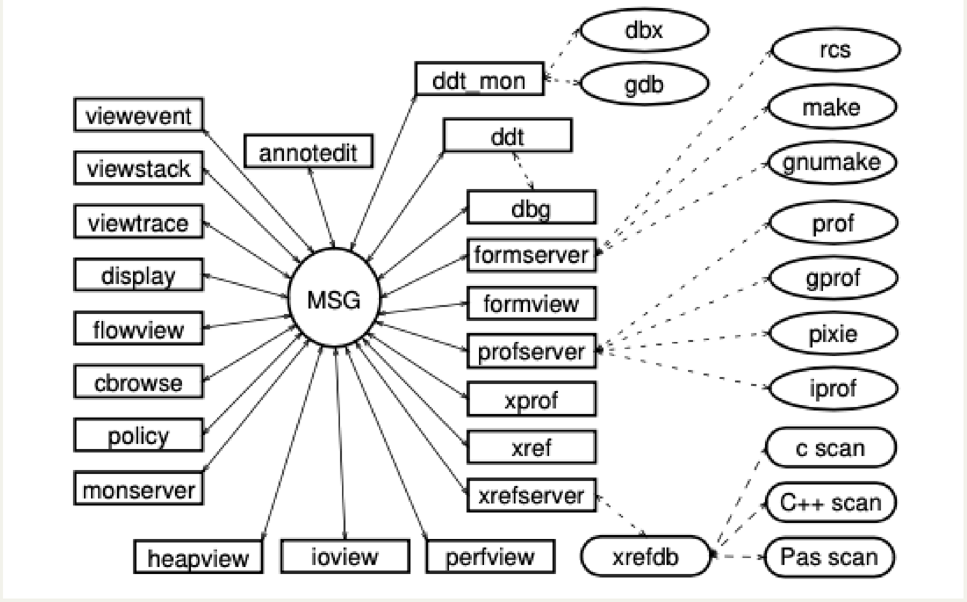


Figure 5.5: The architecture of FIELD

various implementations are available such as the Java Message Service in the Jave Platform Enterprise Edition [SPR: ref], Jakarta [SPR: ref], and Apache ActiveMQ [SPR: ref].

Message-based architectures are a good match for systems that are composed of multiple processes. The messages provide an easy way to coordinate the processes and for processes to make requests or send information to the rest of the system. The set of messages is easy to extend. A message-based architecture allows the loose coupling of various components and make it relatively easy to add components and extend the system.

Message-based systems work best when all the processes are running on a local network. Most distributed systems today include clients that do not satisfy this criteria. For these systems a client-server model is probably more appropriate.

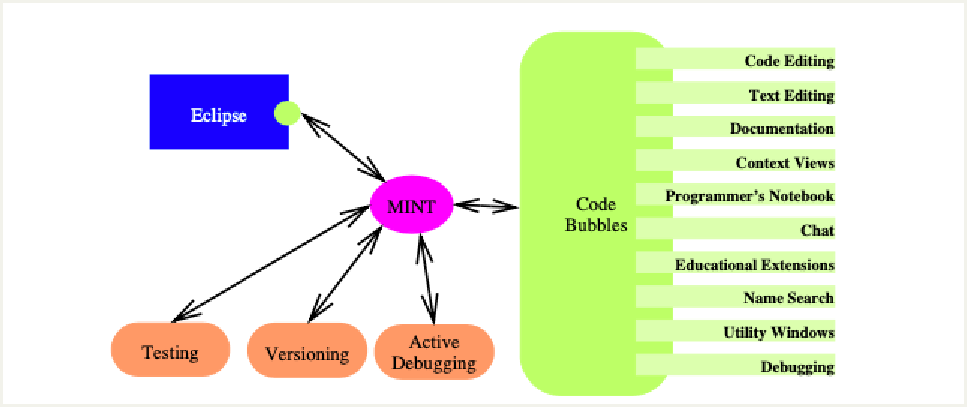


Figure 5.6: The architecture of Code Bubbles

### 5.3.7 Other Software Architectures

This set of software architecture is not meant to be comprehensive. Other architectures exist, primarily for particular domains. For example, real-time and embedded systems will often use a process-control architecture, an architecture based on state machines, or an architecture geared specifically to real time performance. Modern AI systems that use specialized hardware are often architected around that hardware. Peer-to-peer systems, leaf-based computations, and sensor networks provide alternatives for distributed computing for special cases. Robotics programming is architected around the Robot Operating System which provides a specialized message-based architecture **[SPR: cite]**.

## 5.4 Heterogeneous Architectures

Few complex systems follow a pure single software architecture. As Kruchten noted in his definition, software architecture can be defined hierarchically with one architecture multiple architectures being used for different aspects of the system. A hierarchical decomposition might not provide the best high-level view of the system. Instead an architecture that combines elements of the different architectures cited above might provide a more logical view of how a system works.

For example, the Code Bubbles integrated development environment embodies several of the above architectures as shown in Figure 5.6. Code Bubbles is primarily a front end that attempts to make the developer more efficient by allowing an entire working set to be visible at once and providing good browsing and navigating tools. The heart of Code Bubbles is a layered system with a hub and spoke design. The inner most layer provides general facilities such as properties, setup, logging, thread pools, and images. The next layer provides the basic windowing and supports the notion of a bubble which is used as the basis for the displays. The next layer is a browsing framework that tracks all names in a hierarchical fashion. These include names for source objects, names for documentation, names for messaging, and names for debugging. On top of this are the spoke components. These include a component for displaying and editing code fragments, a component for text editing, a component for viewing documentation, a component for browsing the name database, and a component for debugging.

Code Bubbles uses the Eclipse IDE to provide most of the low-level functionality needed in a programming environment. It does this using a plug-in module for Eclipse that communicates with the main system using a message server. Additional back ends for JavaScript and Dart are also supported using the same set of messages.

The message-based framework is also used to support additional processes. One such process provides a testing framework. There is also a spoke component that provides a user interface for testing. Another process provides ties to various version management systems. It too has a spoke-based interface. Another process monitors programs being debugging and provides additional information as they are run, for example profiling data and detailed thread states. It is controlled and used by the debugger spoke component.

Finally Code Bubbles provides a plug-in environment where other components can be added to the system at run time. This is currently used to provide a fast, always-available flow analysis for detecting security problems, a live programming facility, the ROSE program repair facility mentioned earlier, and a UML editor based on UMLet **[SPR: cite]**.

## 5.5 Choosing an Architecture

Choosing an appropriate software architecture is the first step toward developing a comprehensive design for a software system. To do this, one needs to understand what the system needs to do, how it might work, and what other constraints are being imposed on the solution. This is information that can be derived from the specifications. Once an architecture is chosen and used, it is difficult to change.

Hence it is important to determine the proper software architecture early in development and to find an architecture that can work with the eventual system, not just the first approximation. This is why we prefer a lollipop or balloon model of software development where one develops full specifications and then chooses an architecture based on these.

The first step to choosing a software architecture is to decide on a set of basic, high-level components based on the specifications and your understanding of what the system needs to do and how it might work. This should include components not only for the required features, but should also look at the optional extensions that might be needed in the future. **[SPR: Possibly emphasize data over UI]**

Next you need to understand the various software architectural patterns and decide which one or ones are appropriate for your components. Does your system involve phased computations so that a streaming model is best? Is it a web application based primarily on a database? Will it involve multiple processes so you need some type of client-server or message-based structure? Are you building a collection of applications whereby a interpreter-based architecture would be best. If none of these apply then for most systems the default will be a layered architecture. In this case you need to determine what are the appropriate set of layers and how the various identified components might fit into the framework.

In doing this you need to take into account the various constraints identified during specifications. These include the fact that the software will be developed in stages, with a core system being created first. They also include the fact that the software will be developed by a team of programmers and you want to be able to keep the whole team productive during development.

You should also take into account the risks you identified during the specifications process. Ensure that each risk is either isolated in a single component or that, when the risk is major, that the overall architecture is designed to minimize that risk. If you did prototyping, then the architecture should take into account the lessons learned from the prototypes as to what works and what does not.

Finally, you should decide on a simple, consistent architecture that satisfies all these constraints, that will let you build a robust initial core system, and then let you extend that core with any other functionality that has been identified. The key here is to choose the pattern that will simplify the development, both immediately and in the future.

While this sounds complex, it is generally not that difficult. Most often, once you identify the components and understand what the system has to do, the appropriate software architecture becomes obvious. As designers get more experience with software systems, either by developing their own or by simply reading and understanding how systems are built, they get a better intuition as to what should be done for the particular case they are looking at. You can also look at several alternatives and get an understanding as to which is best.

## 5.6 SHORE Software Architecture

There are several basic components that can be identified from the SHORE specifications. The system needs to keep track of what is going on with the rail layout, understanding the state of each sensor, switch, and signal as well as each active train. For this, we assume a *Model* component that provides a model of the train layout and its current state. Next, the system needs to communicate with the engines and the controller for the switches, signals, and sensors. This is done using UDP messaging along with mDNS discovery. For this we assume a *Network* component.

The system need to ensure that switches are set correctly to avoid derailment, that signals are set appropriately, that signals are obeyed, and that rail blocks only contain one train at a time. To do all this we assume a *Safety* component that takes input from the model and sets things appropriately. The system also needs to provide a user interface based on an abstract layout. For this we assume a *View* component.

Extensions to the basic system include using a camera and computer vision to provide additional information about train positions, and the ability to plan and define routes that can be automatically executed. For these we envision a *Vision* component and a *Planning* component respectively.

There are several architectural possibilities. The system is too small to have any of these components be a separate process, so message-based architecture and client-server architectures are not appropriate, even if they could be used and made to work. The Model component is central to the design and one could imagine an implementation where all the other components either modify the model or are triggered by callbacks when the model is updated. This would yield a repository architecture. It is not the best choice in this case because the interaction between the model and networking can be confusing since requests sent by networking may or may not be reported back to the model; and because user interface requests, for example, to trigger a switch, should be passed through the Safety module to ensure they maintain the safety constraints of the layout.

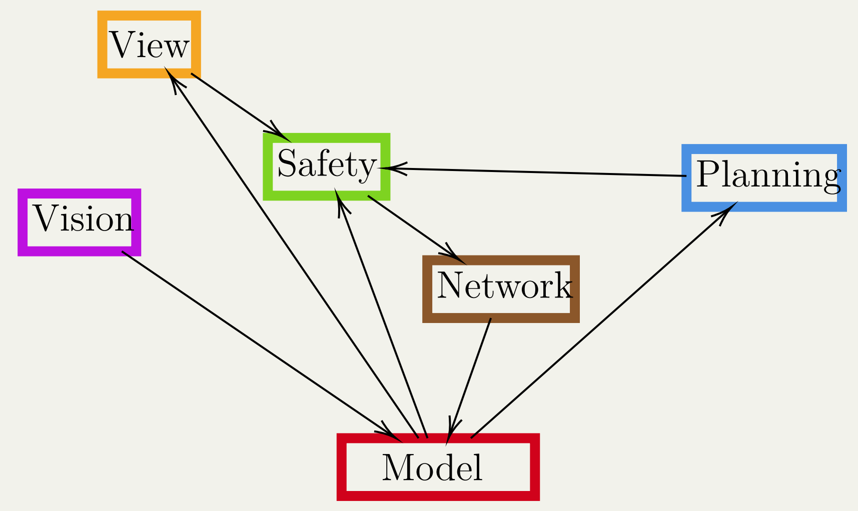


Figure 5.7: The architecture of Shore

Instead, we developed a layered architecture as can be seen in Figure 5.7. The inner most layer contains the Model component. Above this is the Network component which can access and update the model to send and receive commands from the engines and controllers. Above the Network component is the Safety component. It will access or get callbacks from the model to detect when things need to be changed, and will call the networking component to make the changes either to the layout or the trains. The next layer is the View component providing a user interface. The user interface is created and updated from the model. Commands in the user interface are passed through the Safety component and on to networking.

The remaining components, Planning to handle automatic control and Vision to handle enhanced positional information, are viewed as extensions or spokes to this layered design. We need to ensure that the model and train components have enough information to enable the planner to construct and execute plans. We need to ensure that the model can be extended with virtual sensors that can be triggered by the Vision module and that the rest of the system would then work correctly. Both of these conditions are consistent with the layered architecture but will need to be considered in more detail as we get to the detailed design and implementation.

## 5.7 Summary

summary

## 5.8 Further Reading

refs

## 5.9 Exercises

Consider a browser. What are the basic components. How are they connected. What are the key technical problems. How to take advantage of multiple cores. How might the components be organized. What is the appropriate software architecture.

Choose an architecture for your team project. Justify your decision.

# 6. High-Level Design

Once the architecture for a software system is understood, it is time to do the high-level design. High-level design is the bridge between the software architecture and lower-level design of sets of classes or modules. It converts the software architecture into something concrete that can then be designed in detail and implemented.

The purpose of high-level design is to develop a well-defined set of interfaces describing components that can be built by your programming team. Well defined in this case means that each interface provides the details necessary for creating and appropriate implementation and for other interface implementations to use this interface. These interfaces are the end result of high level design and form the basis for the detailed design.

High-level design needs to take into account a variety of factors. It needs to consider the specifications, both those that are specific to the initial system and those that deal with possible future extensions of the system. It needs to handle the various scenarios that were created for the application. It needs to deal with evolving the system and with potential risks. It needs to deal with the security, privacy and ethical concerns noted in the specifications. It needs to be in sync with the number and quality of people on your programming team.

## 6.1 High Level Design

The software architecture for a system defines the set of high-level components. These components are specified only in general terms, for example, by a name and a short description of what they do or what they are responsible for. High-level design takes each of these components and gives them a precise definition in the form of one or more interface. The end goal of high-level is a set of interfaces that can be organized into components so that each component can be implemented in a single package or module by a single developer.

### 6.1.1 Results from High Level Design

High level design yields a set of components that are designed to be implemented by individual developers. Each components is defined by one or more interfaces that specify what information they need to take in, what operations they do, and what information they make available to other components. Where multiple interfaces are used, they are typically highly interrelated.

The interfaces can be specified in various ways. It can be an actual interface or abstract class defined in the target language, for example a Java interface. It can be a language-independent description of a set of calls, for example using a UML class diagram. It can be a set of messages that are to be passed over a socket in a message-based architecture. It can be a set of RESTful calls to be made over the web to a web server. It can be a set of commands that can be issued from the command line. In any case, the interface defines the data that is provided by the component, the data that is provided to the component, and the operations the component needs to perform for other components. All data required or provided by an interface should either be primitive or described by other interface. Each operation of the interface should include a description, either formal or informal, of what the operation does as well as the inputs to that operation and any constraints derived from the specifications or architecture. The information needed from other components by this component is encoded in the interfaces of the other components and the inputs to these operations.

The components described by the software architecture can be quite abstract. They can be a single process in a multiple process system; they can be a single subsystem eventually implemented as a set of packages or modules; they can be a single package or modules containing a set of classes or functions. If the original component is complex enough to warrant a set of packages or modules or to require multiple developers working independently, then it will typically require multiple interface. Defining these interfaces and how they interact is akin to developing a software architecture, but now doing so for the component rather than the system as a whole. Once components are at a level where each represents something that can be implemented by a single module or package, then the task of creating the appropriate interfaces can begin.

### 6.1.2 Correct versus Incorrect Designs

The purpose of high-level design is to create a feasible and simple design that will work. There is no ideal or perfect design for a system. Indeed, almost any design can be made to work. A working design is not the overall goal. Instead, one wants to create a design that is simple, well-defined, and will enhance system development.

A good initial design can significantly reduce the overall workload, both initially and in the future. For example, it can significantly reduce amount of code required. Our experience with student projects in a software engineering class was that the good designs required about half the code that the poorer designs did (1,500 lines versus 3,000). While advanced developers wont see this big a difference, the difference can still be significant.

A good initial design will also reduce the amount of refactoring and rewriting of the code needed during development. Agile programming suggests that the code be refactored as needed to accommodate newer features. By taking potential features into account early in the design process, these refactorings, which can be expensive and error-prone, can often be reduced or eliminated.

A good design will make it easy to add new features, that is, to evolve the system. By thinking about how the system might need to be extended and abstracting the components being developed appropriately, new features, both those anticipated and those that were not, tend to be simpler to add, affecting only a single component, or at worst, a small number of components.

A good design will also be easier to debug and maintain by being compartmentalized and ensuring that each component knows its responsibilities and actions. It will also be easier to test and eventually deploy.

## 6.2 Objectives for High Level Design

High-level design is crucial to the development and maintenance of a complex software system. The design has to be done with long-term objectives in mind to make the development both initial and during maintenance, efficient and effective.

The first objective is to be able to create an initial working system with minimal effort. This means creating a design that will simplify development and lead to a working system as quickly and cheaply as possible. While this is a worthwhile goal, it has to be tempered by the longer-term needs of the system.

### 6.2.1 Design for Maintenance

The target application will be built incrementally and maintained over time. As we have noted, software development of complex systems is essentially all maintenance. Thus, a primary design objective is to make maintenance easier. This is more important, especially in the long term, than creating an initial system quickly.

Designing for maintenance involves understanding what things are going to change as the system is built. These can be outside factors that the system might have to accommodate or they can be inside factors. Outside factors include such things as the operating system interface, any database interfaces, and even the user interface. It might include what outside packages one wants to use, for example which large language model provider makes the most sense to use. It might include what algorithms are to be used.

Part of specifications involved understanding the environment the system will run in and how that might change. This, along with some thought as to future implementation decisions, serves as a basis for identifying those aspects of the system that might change. Once these are identified, the design solution can take them into account by isolating them in a single implementation component, generally with a single interface. Then only that component needs to be changed when future change is required. Note that this might mean creating a common component, for example, a common database access component, that will be used by the rest of the system. Then, the choice of which database system to use is only dependent on that component and the initial choice can be easily updated if needed.

Design for maintenance also involves making maintenance in general easier. This implies design and coding to simplify finding, isolating, and fixing future problems or bugs. At the design level, this can mean several things.

First, one needs to practice defensive programming. The various interfaces definitions should provide constraints on their input, specifying what inputs are allowed and which are not. They should also specify what happens if invalid values are passed in. This means that error handling and exceptions are built into the interface definitions and are a well defined as part of the interfaces before the implementation is started. Exceptions that are specific to the system should be included in the interface specification as separate classes or entities.

Second, one should include a logging framework as part of the initial design. Each component should be able to log information about what it is doing, what might be going wrong, and how it is being used. It is often difficult to duplicate and isolate bugs from just looking at the software behavior. Moreover, especially when the software system is designed to be somewhat fault tolerant, the behavior might not show any problems when there are actually are some. Both of these can be addressed by a logging framework. As the software evolves, understanding what portions of the software might be too slow, too complex, heavily used, or otherwise problematic and help in determine what features should be implemented next. Again, this can be addressed by logging.

Third, one should keep testability in mind when doing design. Testing is critical to ensuring the system works initially and then continues to work as changes are made, new features added, and the environment changes. As you break the system into interfaces, you need to determine how each of the implementations for the interfaces can be tested in isolation. You also need to determine how to create a design that can be tested separately from the eventual production system.

### 6.2.2 Design for Evolution

Not only is the environment to change after the initial software is written, but the software itself is going to change. New features are going to be added, old features are going to be modified, the user interface is going to need updating, the software will find new uses and old uses will become passe. It is important to design the software to make these changes as easy as possible. Changes that are localized, that do not require modifying or rewriting large sections of the system, are generally easier to make, test, and debug. Moreover, such changes are more likely to be implemented since their commitment is lower.

One of the reasons we encourage doing requirements and specifications for a complete system is to have an understanding of what some of these changes might be. The high-level design for a system should look at all the elements of the specifications and ensure that as many future changes, especially ones that might be considered important or critical for a future version of the system, have a placeholder in the high level design. The placeholder might be an interface that is not going to be implemented right away; it might be a call into a module that will be responsible for the feature that is not implemented in the initial version of the system; it might be a component that can be integrated as a plug-in; it might be a separate process that can be integrated using a message passing framework where the information it might need is readily available.

Designing the system to accommodate these anticipated changes will make the system easier to develop and evolve. It will encourage agile development with fewer needed refactorings; it will mean that adding the features can be done without completely rewriting the system.

### 6.2.3 Design for Risk

In section 4.3 we discussed risk analysis as applied to the software system one is developing. The basic idea was to identify potential risks to the success of the software and use these risks to create a design that will ameliorate them. High-level design provides the opportunity to address these risks.

There are three basic approaches to dealing with risk during the design phase of software development. The first is to ignore it and assume it will go away. This can be used if the risk is not understanding how to do something when you know it can be done and you will either bring someone on to the team with the necessary expertise or will spend the time learning how. It also might be an appropriate response when the risk is out of your control, for example depending on the actions of a competitor or the appearance of a competitive product. However, in general it is not a good way of dealing with risk.

The second alternative is to work on the risk and ensure that it will not be a problem. This generally involves developing a prototype system that solely addresses the risk and provides you with the information needed so that the risk goes away. For example, with the SHORE system, one of the risks was the lack of knowledge about computer vision and how it could be used by the system. To address this, we implemented a very simple prototype that used a camera aimed at a portion of the train layout and moved trains along the tracks manually. We determined how computer vision could be used to identify the layout and where trains currently were. We also determined how to use object identification to recognize the different engines. Knowing how these could be done was sufficient so that we felt comfortable that we could implement the computer vision component successfully when we needed to.

The third alternative, which can be combined with either of the first two, is to isolate the risk so that it is contained in a single implementation component, Then if an alternative implementation is later necessary or if major changes are needed, those changes are confined to a that component and do not involve the remainder of the system. As an example, when we build a web application we implement a single component that is the interface to the database. This lets us change from one database system to another, for example changing from a MySQL implementation to a PostgreSQL implementation. At the extreme, in the twitter application mentioned earlier where we had problems with the performance of the database, we actually replaced the database module that used PostgreSQL with an alternative implementation that used MongoDB, a NoSQL database. This was done without having to change any other code in the system.

### 6.2.4 Design for Security, Privacy and Ethics

Security, privacy and ethical considerations in the software, again identified as part of the specifications, can also be addressed directly in the high-level design using tactics similar to those used to mitigate risks.

Specifications detailed what data needed to be kept secure, what data was sensitive and should be kept private, and what data is company confidential and should not be exposed to users. The goal of high-level design is to ensure that these various constraints can be met by the implementation. The easiest way of doing this is to ensure that each of the identified concerns involves a single implementation component. This allows the component to implement whatever security measures are deemed appropriate such as encryption or detailed access checks without affecting the rest of the system. It also ensures that as the security or privacy constraints change as the system evolves, that such changes are contained in a single component. Using a separate component lets that component implement the various operations on the confidential data without exposing that data.

Ethics is a bit trickier. Ethics involves how the system will eventually be used, and will it be used fairly and evenly, without causing discrimination and without adversely affecting other people. While some of this might be obvious based on the proposed use of the system, much of it involves the future and is not easy or sometimes even possible to predict.

Ethics is becoming more important, especially as systems are being used to make critical decisions, for example, who gets a loan or who gets what health procedures. Much of this is due to the use of artificial intelligence integrated into a system where the behavior or the AI component is not well understood or described. From the design perspective, it is a good idea to design the system so that recommendations or responses can be justified by the system. Explainable AI is getting at this, but designing systems with this in mind to start with should help. Another way of addressing potential ethical considerations, especially elements such as fairness, is to ensure that there are adequate test cases and adequate testing for this. Some of this can be done as part of the high-level design.

### 6.2.5 Design for Team Development

Designing for evolution, maintenance, risk, security, privacy and ethics is somewhat vague because the concerns are often not particularly specific and the approach is mainly one of isolating the affected components so that changes that might be required to address these concerns are isolated in a single implementation component and thus are significantly easier to make. Designing for team development is more concrete.

You have a team of programmers and want to have them work efficiently. You want to have a design that makes team development easy and efficient. This implies that they should be able to work in parallel and independently. This prevents everyone in the project waiting for a slower programmer or a programmer who is working on a particularly difficult aspect of the system. One should arrange the high-level design to facilitate that.

The basic idea is that each team member should have their own code or implementation components. This lets the developers work independently and asynchronously. It also facilitates individual testing and debugging of the components. Note that it is possible (and at times practical) for components to be assigned to a small programmer team, say 2 or 3 people, rather than a single developer. The small team can then divide the component up further so that they can work independently.

Design for team development builds on the interfaces defined for the components. With a well defined set of interface, developers know exactly what they need to code and they also know exactly how to use the other components in the system to get their component to work.

Design for team development also tries to tie the number of implementation components to the size of the team. It tries to ensure that there is a component for each team member to work on at any time. This might mean that an individual developer is assigned multiple, simpler components.

Design for team development also defines when to stop high-level design. The design of an individual component should be left to the individual developer and does not need to be part of the top-level design. As long as there is an interface to the component, it should not matter how that component is actually implemented in terms of subcomponents, classes, and methods.

## 6.3 Approaches to High-Level Design

High-level design starts with the components identified in the software architecture. From these it defines a set of implementation components each of which is defined by a small set of interfaces. This ensures that the design reflects the software architecture. It takes each of the architectural components and creates appropriate interfaces for that component that reflect the corresponding implementation components. While a single interface for each architectural component might be preferred, multiple interfaces are often needed. They are used when the architectural component is too complex to implement in a single package and thus involves multiple implementation components. They are used to represent data structures that are passed in and our of an implementation component. They are used to represent callbacks from an implementation component. They are used to facilitate isolation of portions of the code that might need to be changed as the system evolves.

### 6.3.1 Selecting Implementation Components

The first step in high-level design is to determine what interfaces are needed to cover the components in the software architecture and to ensure these are sufficient for the application.

Ensuring that the overall design works is typically done by looking at the various scenarios developed as part of the requirement specifications and understanding exactly what each of the components needs to do to make that scenario work. This should ensure that the interfaces at least contain the basic elements that are needed for the scenarios. It should also provide a good understanding of what the various interface elements should do.

The implementation components should be selected with respect to the goals cited for the design. Where design elements need to be encapsulated, for example for evolution or for security, there should be a single component that deals with the issue. Where a single component is not possible, for example when a feature needs to be added to both the front end and the back end of a web application, the number of components involved should be kept to a minimum.

The implementation components should also be selected so they can be assigned to an appropriate team member and so that all team members will have something to work on. This might mean taking into account the expertise of the different developers on the team and creating corresponding components.

The implementation components should also represent functionality rather than shared data. Where data is to be shared, it should be shared procedurally through the interface rather that as actual data. This provides the most flexibility in terms of the implementation and allows easier evolution.

### 6.3.2 Interfaces

The key to high-level design is to specify a detailed and relatively complete interface for each of the implementation components. This takes a component from a vague box in a diagram to something that is actually buildable. It provides the detail needed to understand the component; it provides the detail needed for other components to use the component; it provides the detail needed to actually implement the component.

Creating the interface for a component is an attempt to understand precisely what the component needs to do and how it will be used. It is also an opportunity to check the high-level design and architecture and make sure it can work. It provides the chance to revise the design while the cost of doing so remains relatively inexpensive. It lets the designer check if the various components are indeed the right components, if they are the right size, if their interface is coherent and consistent.

This means that when creating the interfaces for a component, one should continually check whether the interface is the correct one. One needs to check that the interface meets the various specifications. One needs to ensure that other components have access to whatever information or functionality they need. One needs to ensure that the functionality implied by the interface can be implemented and that the implementation will be about the right size. One needs to ensure that the various constraints imposed by the specifications and design requirements are met by the interface.

Note that we use the term interface fairly loosely. It can be actual language-based interfaces, say a set of Java interfaces or a set of C++ abstract classes without implementations. It can be a set of function calls, for example offered by a library. It can be a set of messages or HTTP requests. It can be a set of command line options that control the behavior of the subsystem.

Interfaces can also be bi-directional. Often one component will need to inform another component when something happens. To accommodate this, the interface definitions need to include callback classes and the ability to register and unregister for the various callbacks.

Interfaces should also be relatively complete. They should include not just the calls that are available, but descriptions of those calls. They need to include appropriate documentation because simple method or function names, while they can convey much of what the call should accomplish, are never complete or fully explanatory.

Interfaces should also include error handling. In any system implementation, it is always important to understand what can go wrong and what the various components will do when there are problems. If a method can throw an exception, this needs to be included in the interface (as might the exception). The interface should provide constraints on the parameters to calls, for example, that a given parameter is not null or is greater than 0 and less than the length of a string. It should also describe what happens when these constraints are violated.

The interfaces that are created for the design should follow the general guidelines for designs. They should be kept as simple as possible. There should not be a large number of classes. Each class should have a relatively small set of methods, functions, or messages. Class hierarchies should be represented only by the root class or an interface representing the root class. Each method should have a small number of parameters. Constraints should be imposed on individual methods or functions rather than on sequences of calls.

At the same time, the initial interface definitions should provide room for expansion. They should identify possible future classes, methods, and messages. They should identify how the system can easily evolve to handle all the functionality indicated in the specifications. It is okay to have interfaces that will not be implemented right away.

Also note that the interfaces will change. They will change as individuals develop the corresponding implementations and find they need other methods or that they can easily provide other functionality that will be needed. They will change as the specifications change over time.

## 6.4 Representing a High-Level Design

A high-level design needs a concrete representation that can form the basis for the subsequent implementation. This representation should define the various components and their interfaces. It should provide a representation that describes the functionality without describing the implementation or imposing restrictions on the implementation. It should provide the basis for the detailed design and coding that will follow.

Essentially, the high-level design provides and application program interface or API for each of the components. It needs to define the calls and requests that the component will handle, including the various parameters and return values. It should include documentation on what the calls do and what errors can occur. The representation needs to define data formats where data is passed between components. It needs to define any conventions, calling orders, etc. that should be kept consistent among the various components. It also needs to define callbacks, specifying how and when they are used.

This is what is needed. The issue then is how to represent it. There are several alternatives, each with their advantages and disadvantages.

### 6.4.1 UML-Based Descriptions

One approach to representation is to use UML class diagrams to represent the various components. UML class diagrams are a graphical representation of a set of classes along with their methods. It allows classes to be grouped into packages if needed and includes the signatures of the various methods. It can also include links between components that show potential interactions.

A sample UML class diagram is shown in Figure 6.1. A UML class diagram consists of boxes and lines. The boxes represent classes and can be divided into 3 regions. The top region shows the name of the class. The second region shows the set of fields or variables, which UML calls attributes. The third region shows the set of methods or functions, which UML calls operations. The attributes and operations include information about the type or signature as well as visibility. When using class diagrams for high-level design, one is only concerned with public methods or functions and generally not concerned at all with fields.

The lines in the diagram represent various types of dependencies. These can be inclusion or composition, that a particular class includes a component or a set of components of another class; they can indicate dependencies, that one class depends on another class; and they can indicate generalizations which represent a class hierarchy. For high-level design, very little of this information is relevant. Typically, a high-level design would only include the top class of a hierarchy; composition of items is an implementation detail and is generally not included in the design; and dependencies are a result of the implementation, not a component of the design.

Thus a high-level design representation with UML generally consists of a set of

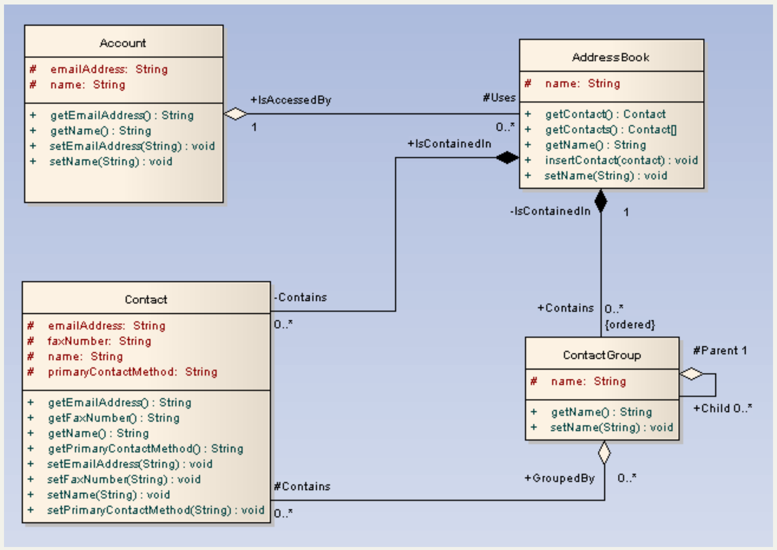


Figure 6.1: Sample UML Class Diagram [SPR: replace]

boxes with a label each of which represents a component and then with a set of associated operations which define the interface to the component. While attributes are not needed to describe interfaces, they can be helpful in understanding the various components.

UML class diagrams have the advantage of being graphical and somewhat easy to create and manipulate, especially in the early stages. They create a nice, compact visual representation of the system. However, we have not found a UML editor that is particularly user friendly. Moreover, UML diagrams include many nuances that are not particularly relevant to the high-level design. They are also a step away from the eventual implementation which makes them less concrete. As a separate abstraction, they are unlikely to be kept up-to-date as the system evolves.

Because of these reasons, we prefer to use UML diagram or something like them as a first step, essentially for identifying the implementation components, but not defining them in detail. For example, for SHORE, we created an initial UML class diagram showing what we saw as the implementation components as seen in Figure 6.2. This diagram is quite similar to the software architecture for SHORE seen in Figure 5.7. We have create a top level implementation interface for each of the architectural components. We called these factories since they are responsible for creating the

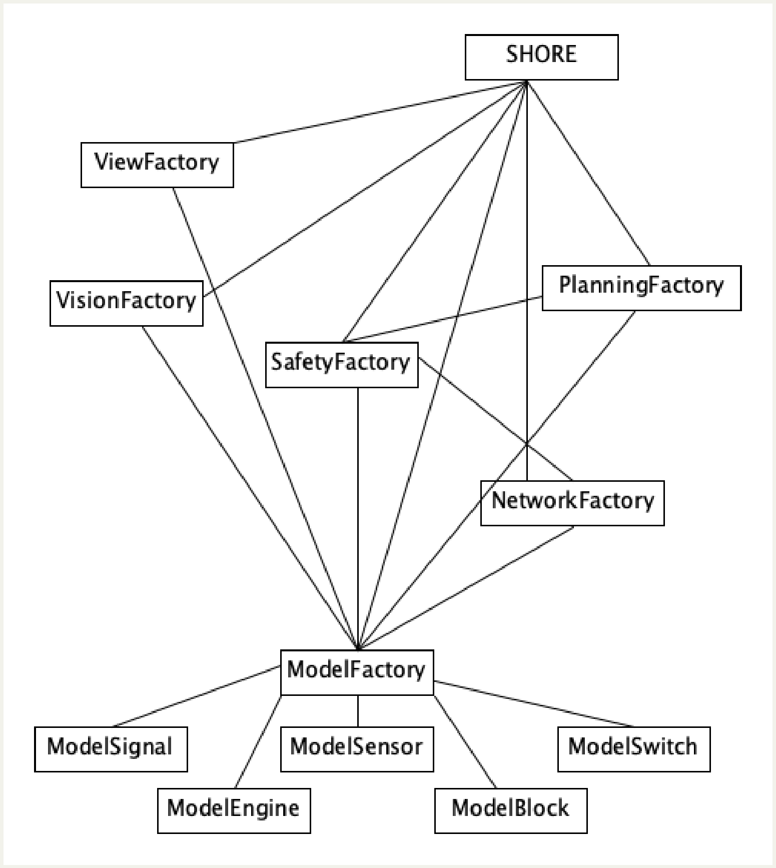


Figure 6.2: SHORE Implementation Components

actual implementations and provide a facade for the component. For the model component, we added interfaces for the various model elements that are identified in the specifications. We also added a SHORE component that represents the main program and is responsible for setting up the system.

### 6.4.2 Language-Based Descriptions

A more concrete description of the high-level design can be developed by creating actual interfaces in the target language. Such a representation provides a concrete definition of the signature of the various operations and provides a means for documentation. It serves as a basis for the future implementation. There are two primary ways of doing this. **[SPR: Might want future references to chapter 10 on choosing the language and chapter 12 on coding style as a lead in to language-based design.]**

The first involves creating all the initial interfaces in a separate package or module and then having the various component implementations in their own modules. Each module then references the interface module to get the information needed to access any other modules. Each implementation module also provides a concrete implementation of one or more of the initial interfaces.

Language-Based interface definitions are created incrementally. The developer creates initial interfaces for components based on the software architecture or a UML-based representation. Then they add methods that seem to be required. For each method, they note what it requires and what it returns. When it requires information from other interfaces, the corresponding methods can be added. Methods can be removed if they are duplicated or not needed; methods can be combined as needed. The goal is to create a simple representation containing just the essentials for each component. Since one is only working with interfaces and abstract methods, things are easy to change. The eventual interface package then provides a concrete view of the high-level design.

As an example, Figure 6.3 shows some of the interfaces defined initially for the SHORE model component. The top level interface defines the model and provides access methods to provide the set of sensors, switches, signals, blocks and engines. It also provides a callback interface with methods that will be called if the various elements change state as well as methods to add and remove callbacks. Then it includes interfaces for the various model elements as well as enumerations defining what states these elements can be in. The interface would also include JavaDoc comments describing each of the methods, enumerations, and interfaces.

While interface-based definitions are convenient and a solid foundation for creating the actual implementation, they do have disadvantages. They require a commitment by the developers to the target language and to coding conventions. They are not the best match for message-based or RESTful architectures, although they can be used where by implementing a single module that makes the RESTful calls.

They also tend to make the eventual implementation a bit messier. They might require more trivial classes and public methods. The implementations will often have to cast an argument from its interface to its internal representation. Collections of items from the interface cannot always be created from collections of implementation items.

```
public interface ShoreModel {

Collection<ModelSwitch> getSwitches();
Collection<ModelSignal> getSignals();
Collection<ModelSensor> getSensors();
Collection<ModelBlock> getBlocks();
Collection<ModelEngine> getEngines();

void addModelCallback(ModelCallback cb);
void removeModelCallback(ModelCallback cb);

interface ModelCallback {
   void sensorChanged(ModelSensor s);
   void switchChanged(ModelSwitch s);
   void signalChanged(ModelSignal s);
   void blockChanged(ModelBlock b);
   void engineChanged(ModelEngine e);
}

enum ModelSensorState { OFF, ON, UNKNOWN }

interface ModelSensor {
   ModelBlock getBlock();
   ModelSensorState getSensorState();
}

enum ModelSwitchState { N, R, UNKNOWN }

interface ModelSwitch {
  ModelSensor getNSensor();
  ModelSensor getRSensor();
  void setSwitch(ModelSwitchState s);
  ModelSwitchState getSwitchState();
}

enum ModelSignalState { OFF, GREEN, YELLOW, RED }
interface ModelSignal { ... }
enum ModelBlockState { EMPTY, INUSE, UNKNOWN }
interface ModelBlock { ... }
interface ModelEngine { ... }

}        // end of interface ShoreModel
```

Figure 6.3: SHORE Model Interface Excerpts

Evolving the system by adding methods to an interface that multiple components implement, for example a callback interface, might either require changes in multiple parts of the system or require the definition of an additional interface and then having the implementation code check if the actual object passed in implements only the original interface or the new interface as well. Changing an interface once the implementation exists by anything other than adding methods might require changes in multiple components. The interfaces will grow as the system evolves and the eventual result, viewed as a design might seem overly complex, containing implementation details that are not necessary for the design.

An alternative language-based implementation is to create a facade-based design. Here, rather than creating a set of interfaces for a high-level component, one creates a facade class. This means creating the package or module for the implementation

```
package edu.brown.cs.shore.view;

import edu.brown.cs.shore.ifaces.ShoreModel;

public class ViewFactory

public ViewFactory(ShoreModel model)
{ }

public void startDisplay()
{ }

}        // end of interface ShoreModel
```

Figure 6.4: SHORE View Component Facade

component and then populating that package or module with a small set of classes or interfaces that have the appropriate public methods but no real functionality. Ideally this facade is a single class or file that serves as the public interface to the component.

For example, Code Bubbles was implemented using a facade-based design. Its core consists of a small set of classes (under 20) in the key packages. There were empty classes in the control package Board for logging, properties, and setup. There were empty classes in the basic windowing package Buda for the bubble area, bubbles, and bubble groups. There was one class in the back end communication package Bump representing the back end. There were classes in the explorer package representing names and repositories. There was a single class in the editor package that provide a facade for creating and accessing editors, and a single class in the bubble stack package for creating bubble stacks. This type of design was then used as Code Bubbles evolved. New packages, as they were created were implemented using a single facade class that set up the package, defined any menus and bubbles, and provided access to the facilities of the package.

While we implemented SHORE using an interface-based design, we could have used a facade based design. If we had, rather than creating an interface representing each component we would have created a facade class in the implementation package for each component. For example, the view component would have been represented by a class similar to that of Figure 6.4. This class references the model which is the basis for creating and updating the viewer. It includes a constructor and a single method that starts the display.

Facade-based design is a good match for a layered system where each facade represents a layer. They are also a good match for extensible component systems where new components are added using a standard interface. Here the facade class simply implements that interface and new packages can be added relatively easily. For example, Code Bubbles used this approach to provide the various extensions, for example for testing and version management. Here each new package is defined using a default public constructor and then two optional public classes for initialization, one called before windowing is set up and one called afterwards.

Facade-based design can also be slightly easier to use that interface-based design since it does not require any odd factory methods or classes for creating instances of the various interfaces and it is easier to add new components and to evolve components. However, it also has its disadvantages.

Facade-based design can make it more difficult to implement the components in parallel. The inner layers need to be defined before the outer layers can be written. Moreover, it is easier and hence more tempting for the developer to change the interface between their component and others since the interface is embedded in the package or module rather than in a separate package or module. While interface-based design provides a central description of the overall system in the package containing the interfaces, facade-based design provides only a distributed description which can be more difficult to understand and get the overall gestalt while creating the high-level design. There is also a tendency to avoid documentation, especially early in the development process.

### 6.4.3 Message-Based Descriptions

Language-based designs can be appropriate and simple when creating an application in a single target language where interaction is done through what are effectively function calls. They are less appropriate if one is creating an application that uses multiple languages and where the communication is done using a messaging framework. In particular, this applies to web applications where there is a front end written in JavaScript or Typescript running in each users browser, and a back end that can be written in a variety of languages running on a server in the cloud.

If your software architecture is message-based then the high-level design should reflect that. In this case the architecture should identify the messaging framework. For web applications this is a set of HTTP requests, generally ones following a RESTful protocol. At this point you should determine the overall format of the messages. For example, a RESTful set of messages generally have a prefix denoting what aspect of the system is being accessed, a term indicating what type of item is being accessed and the data for that access. The XML-based messages used by Code Bubbles generally start with the source of the message and then include a command parameter, other parameters to identify the particular instance, and parameters with the command arguments.

Once you have determined the overall format of the messages your application will use you should enumerate the set of messages that will be needed, at least those that will be needed in the initial system. Since messages are quite flexible, it generally is not necessary to define messages for all likely extensions, although doing so to ensure the extensions fit in logically with the rest of the system does not hurt.

For each message you should specify the format, the arguments, the expected behavior, the resultant reply or messages that will be returned, and any error behaviors that are appropriate. You should also note what should be done if the message was not received, i.e. if there was not acknowledgement for the message. This should be akin to defining the appropriate set of function calls in a module or interface. These should be kept in a global file and should be maintained as the system evolves. The important point is that this file defines the high level interface between the components and should be treated as an interface.

Message-based interfaces and their implementations have their advantages and disadvantages. They are relatively easy to change, augment and extend. They are quite flexible, allowing additional optional arguments to be added on the fly. They are the best fit for web applications where work is to be divided between the front and back ends. They are a good fit for applications that involve multiple processes, whether on the same machine or distributed across a network.

On the other hand, they are generally not appropriate for interfaces between two component in the same process, although this can be made to work. For example, the message server used by Code Bubbles will work with components both in separate processes and in the same process. Message-based interfaces also can make concurrency and synchronization more difficult. The applications needs to determine if it can handle multiple messages simultaneously and then provide the appropriate synchronization. The front end needs to know that messages might not get an immediate reply and might need to allow processing, for example the user interface, to continue while waiting for the message.

There is also a tendency when using a message-based interface to avoid creating documentation as new messages are added to the system, and updating the documentation as messages are changed. This is because the document describing all the messages is not code as it would be in a language-based description, but a separate file and the messages themselves are relatively easy to change.

## 6.5 Summary

summary

## 6.6 Further Reading

refs

## 6.7 Exercises

Create a high-level interface-based design for your project. Do an initial version as a UML class diagram. Do an actual version as a language-based description and, if appropriate, include detailed documentation on any message-based interfaces.

Take specifications for your simple project (from chapter 4) and create an interface-based design based on them.

# 7. Design Issues

Modern applications are rarely built from scratch. Instead, they are built around or on top of other tools and systems. This creates its own set of constraints that influence and affect the top level design and that need to be understood as part of the design process.

Similarly, modern applications are typically concurrent applications, with multiple processes and multiple threads of control. The developer needs to understand the implications of this for design and the further implications for debugging and testing that can be affected by the design.

In many applications, especially brand new ones that attempt to tackle a problem that has not been addressed before or that attempt to take a new approach to a problem, there are a number of unknowns what arise in the specifications as risks. Prototyping provides a way of understanding how to design such systems so that the risk is minimized.

How these issues affect design, and especially high-level design, are discussed in this chapter. **[SPR: Need more figures to break up the text.]**

## 7.1 Designing Around What Exists

One of the goals of software development is to keep things as simple as possible, both for the immediate development and during maintenance. One way of doing this is to avoid writing code at all and use the work of others. A large amount of software has been written and made available in various forms. In many cases it does not make sense to build everything from scratch. Rather one should reuse the work of others when practical.

When practical in this context needs to consider a variety of factors. Using outside code should simplify your efforts, both immediately and in the future. If its more work to use outside code, then it is probably not worth doing so. You also have to weigh the costs, benefits ad risks associated with using outside code.

The benefits are generally obvious. There is a lot of standardized code available. These can be in the form of libraries that have evolved over time and in common subsystems that are widely used. Many of these libraries and systems are actively maintained by others. Moreover, they are typically, but not always, well-debugged and tested. They have a well-thought out application program interface (API) that has been tuned to a wide variety of applications and is often able to meet both current and future needs.

The costs and risks are not as obvious. There is a learning curve to using an external package, which will vary based on the package, the application, and the documentation available. Intellectual property rights need to be considered. If you plan to use open source software you should consider the license it is offered under and whether the terms are compatible with your application.

There is a risk that the external software will no longer be maintained. A while ago I was using JikesBT from IBM for Java byte code manipulation. This was helpful as IBM maintained the software, updating it when a new version of Java was available. However, IBM dropped maintenance as of Java 1.5. I tried maintaining the package myself for 1.5, but handling Java 1.6 was overly complex so I ended up having to convert all my code to use a different byte code manipulation package, which took considerable effort.

There is also a risk that the software will have problems. Debugging foreign code can be difficult and might not be something your project team wants to handle. You should look carefully at any package you are thinking of using and ensure that it is actively maintained and widely used. The software also should be compatible with your software architecture and execution environment. Considerations here include how much memory and CPU it might require, what file resources are needed, does it run as a separate process. Is it compatible with all the platforms you intend your software to be available on.

There are a number of well-understood domains where it almost always makes sense to build on top of existing systems. These reflect aspects of programming that occur in multiple applications, and include databases, text search, and machine learning, among others. In general, in these domains it is best to use existing code, even if you particular application in the domain seems quite simple. Typically these existing systems involve complex, tricky implementations with significant error checking, which might seem like overkill for a simple use, but will be helpful as the system evolves. The tools are designed for complex applications and, while your initial use might be simple, software only gets more complex with time and you are likely to need some of the advanced features in the future.

### 7.1.1 Database Management Systems

Most applications require reading or writing information from disk. This can be simple information, for example configuration data, or it can be complex information, for example the current inventory for an on-line store. How your application handles this depends on the particular type of data, its complexity, its importance to the application, whether it can be edited directly by the user, its need for privacy and security, and how the application will evolve.

In general, it is a bad idea to write your own routines for inputting and particularly outputting application data, especially when the data is application critical. A lot of things can go wrong when writing and storing data. The program can abort or be terminated in the middle of a write; the disk can get corrupted; your program might try to write the same data at the same time in different locations and corrupt the data; the user might edit or delete the file in some way even if that was not the intention. Moreover, the size and complexity of the data is only going to increase as the system evolves and your routines will then have to evolve accordingly.

Where the data is designed to be edited by the user, for example configuration information used by the program, you should use standard file formats and the available libraries for reading and writing in those formats. For example, Java provides a properties class that offers the application a map-like interface for accessing properties but provides methods both to read and to write the current set of properties from or to disk in either line-oriented form or XML. Similar libraries are available for other types of property files, for example YAML. For other data, it is best to write it out in a readable form, using either XML or JSON. Again, most languages provide libraries for doing this.

When the data is more complex, privacy or security is a concern, you need to avoid corrupting the data to maintain the application, and you need to access the data either from multiple parts of your application or from multiple processes in your application, the preferred solution is to use a database system.

There are two basic types of database system, relational system that store data in tables and use SQL as the interface language, and No-SQL systems that are primarily key-values stores but can store arbitrary data in a format such as JSON. Both of these come in a variety of forms. Both types of systems will provide your application with reliable, permanent storage. Before choosing a system for your application you should understand the pros and cons of the different types and decide which is best suited to the application's current and future needs.

Database systems can be used effectively for both long-term and short-term storage of data. They offer data that is difficult to corrupt and can provide redundancy to maintain the data even with system or disk failures. They can handle data coming and going to multiple processes. They can be distributed to offer additional redundancy, handle large volumes of data, and provide a boost in performance.

In general, using a database is not slower than doing something specific for your system. The database generally runs in a separate process and communicates with your application using a socket. Writing to the socket is generally faster than writing to disk. Database systems are heavily optimized, and cache data as needed. While writing data to disk, they attempt to keep the data that is needed in memory for better performance. They do extensive error checking and handling to avoid any data corruption.

There are a host of SQL database systems available. Some, like SQLite, Microsoft Access, and Derby, are designed for a single non-distributed application. Others such as MySQL and PostgreSQL are open source, freely available and quite robust. If you need more capabilities or support, there are commercial systems such as Oracle, IBM Db2, and Microsoft SQL Server. They all share the same basic interface between the application and the database, so it is relatively easy to migrate from one to another if needed. However, the interfaces are not completely standardized and the different systems have slightly different syntax and approaches for more complex operations. For example, one simple difference is how arguments are defined in a prepared query. For this reason, if you want to use a SQL database, you probably should define a component as part of the high-level design that will manage access.

Similarly there are a variety of No-SQL systems available. These tend to have different interfaces and migrating from one to another is more complex. The most common one to use is MongoDB which has a free version as well as fully supported, more robust and powerful versions.

While using a database system in your application offers significant advantages, there are other considerations. They provide an additional point of failure and an addition component that is going to evolve possibly independently from your code. If you plan on porting your application to other platforms you need to ensure that the database system can also be ported. If you plan on distributing the portion of your application that uses the database to users, then you have to either distribute it with the database, or assume the user has a database and then you have to get credentials to use their database.

### 7.1.2 Other Standard Packages

Databases are not the only domain in which there is existing, well-developed, debugged, maintained, and available code that you will probably want to use in your project if your application requires it. There are several domains that occur over and over again where it does not make sense to write your own code.

At the simple level, there are basic data structures and algorithms. Today, most of these are built-in to the programming language, either directly or as easily accessible libraries. For example, Java include a variety of list, set and map data structures including ones that are safe for concurrent use, as well as basic algorithms for sorting and searching these structures. Whenever possible, you want to use the built-in libraries. If you need a more complex data structure, it is also best to build it on top of these libraries rather than from scratch.

Beyond this, there are specific domains that can be very nuanced where a lot of thought and effort has been put into creating usable and useful libraries. These libraries are generally well understood, well documented, and widely used. These domains include computer vision, machine learning, and linear algebra.

Computer vision involves a variety of algorithms and data structures. The OpenCV **[SPR: cite]** library includes most of what an application would need if you are going to do anything with vision. While most of the documentation and usage involves using Python scripts, the library is actually easily accessible from a variety of languages. This is what we have experimented with and will be using in SHORE for example.

Machine learning is both a well-understood and an evolving area. There are a number of standard machine learning algorithms that can be applied to data including those using Bayes theorems, those that estimate a functions, those that look at nearest neighbors, those that use trees, and combinations of these. Standard implementations of these exist, and, if your application needs to do simple machine learning, should be used rather than attempting to implement your own. I have used the WEKA package **[SPR: cite]** for this purpose quite successfully.

Today's machine learning, however, is more complex. Large-language models (LLMs) have shown significant ability to do learning and applied artificial intelligence. There are currently several competing LLMs such as ChatGPT, LLAMA, Claude, and BERT. While they all do about the same things, the interfaces for using them, for augmenting them, and for incorporating them into an application are different. Still, using one of these pretrained models as a starting point is much easier than attempting to develop ones own LLM.

However, one should note that the LLM landscape is not yet stable and that the set of LLMs is likely to change over time. Moreover, the interfaces for using them and incorporating them into an application is also going to undergo rapid changes. If you plan to use a LLM in your application, you should plan on needing or wanting to change which you use and on the interface for using it changing.

Another common domain, especially in applications that do any type of scientific computing, is linear algebra or matrix manipulation. The algorithms needed here vary considerably depending on the type and nature of the matrices involved. Moreover, the algorithms can be quite nuanced, having to handle round off errors and various infinities. Again, if your application needs to work in these domains it is better to use a standard library, for example LINPACK **[SPR: cite]**, rather than developing your own.

Another library that many modern applications might want is text search. If the application has a large manual or set of help pages, or it involves a large set of web pages, it might want to provide users with the ability to search for a set of keywords and find relevant locations in the manual or pages. This is generally done by creating an inverted index.

The algorithms, data storage, and various query techniques for inverted indices can get somewhat complex. Again, it is not worthwhile for your application to try to implement these algorithms directly as existing libraries, such as Apache Lucene **[SPR: cite]**, have already done so. Lucene provides a very flexible interface and can be adapted to a wide variety of applications. Alternatives in this domain exist. If you only need to search web pages, for example, your documentation is on-line, you can embed Google search into your application. This provides a simple high quality search, but is not free and open source as Lucene is.

Other domains where your application should use existing code rather than attempting to develop their own include cryptography where the algorithms are notoriously difficult to get right; web scraping where standard libraries exist based on CSS-type queries; editors which can be quite complex to write but where various frameworks for such editors exist both within the standard library and as easily adapted addons.

For example, Code Bubbles adapted the standard Java editor framework to provide a very sophisticated code editor and then used the jsyntaxpane extensions **[SPR: cite]** to provide basic editors for a variety of file types. For web applications, there are a wide variety of readily available plug-ins both for the front-end where jQuery used to be the standard, and for the back end, for example the various extensions provided by NPM for Node.js applications.

### 7.1.3 Open-Source Code

While it is tempting to use as much external code as possible, one should be aware of the costs and risks as well as the benefits of doing so.

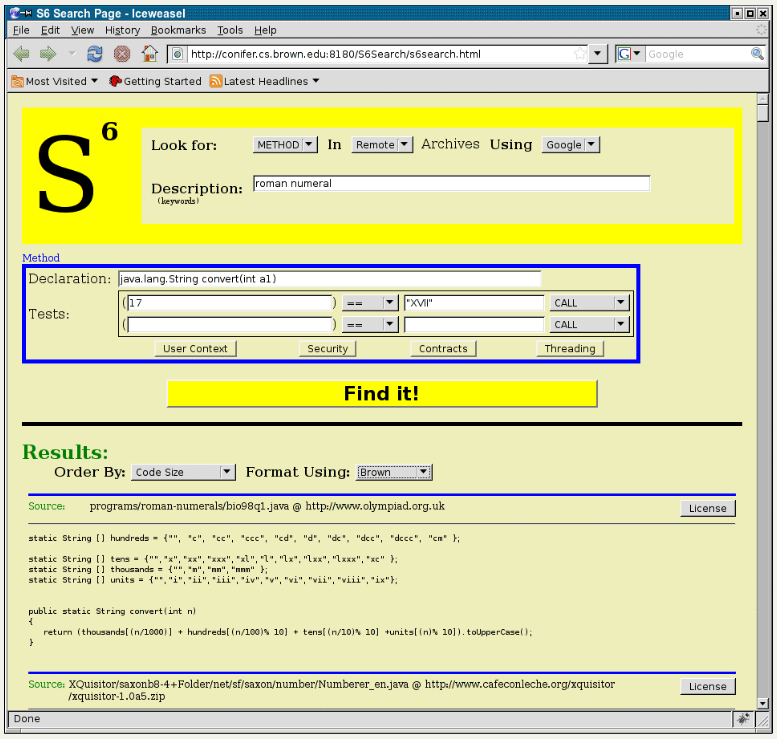
For any library, there are both costs and risks. There is a cost to learn the library, especially if no one on your project team has used it before. This is especially true if the documentation for the library is not complete or descriptive enough, as is often the case. There is a cost to maintaining your code as the library evolves. Most of the libraries cited above are fairly stable, but other libraries can change significantly over time.

There is a cost to adapt the library code to your particular application. The library might make certain assumptions about the environment it is to be used in. While these are generally flexible, you should be sure that your particular use fits into their framework. If you are not planning on using the framework as a library, but are instead planning on incorporating the code directly into your application, then there is a cost of adapting the external code to your coding and usage style.

Some libraries might not be compatible with your current plans or might require committing your whole application to using them rather than just the components that need them. For example, the Apache collections library provides an extension of Java Collections, but if they are used in one portion of the system, then you have to be very careful to ensure that other portions of the system can just use the standard Java calls.

Libraries also have their own dependencies which might conflict with your design and plans for the system. For example, many libraries use the Apache Logging framework, which might or might not be the your preferred logger. Libraries tend to also pull in other libraries, especially with Maven making this relatively easy to do. If two different libraries try to use different versions of a third library, your system might become unstable.

Finally, you want to ensure that attempting to use an external library does not skew


S6 Search Page - Iceweasel
File Edit View History Bookmarks Tools Help
http://conifer.cs.brown.edu:8180/S6Search/s6search.html
Google
Most Visited
Getting Started
Latest Headlines
S 6
User Context
Security
Contracts
Threading
Find it!
License
Done

Look for: METHOD In Remote Archives Using Google

Description: (keywords) roman numeral

Method

Declaration: java.lang.String convert(int a1)

Tests: (17) == "XVII" CALL

( ) == CALL

Results:

Order By: Code Size Format Using: Brown

Source: programs/roman-numerals/bio98q1.java @ http://www.olympiad.org.uk

```
static String [] hundreds = {"", "c", "cc", "ccc", "cd", "d", "dc", "dcc", "dccc", "cm" };

static String [] tens = {"","x","xx","xxx","xl","l","lx","lxx","lxxx","xc" };
static String [] thousands = {"","m","mm","mmm" };
static String [] units = {"","i","ii","iii","iv","v","vi","vii","viii","ix"};


public static String convert(int n)
{
   return (thousands[(n/1000)] + hundreds[(n/100)% 10] + tens[(n/10)% 10] +units[(n)% 10]).toUpperCase();
}
```

Source: XQuisitor/saxonb8-4+Folder/net/sf/saxon/number/Numberer_en.java @ http://www.cafeconleche.org/xquisitor /xquisitor-1.0a5.zip

Figure 7.1: S6 Display Showing Input and Output

or disrupt an otherwise clean design, thus making future maintenance of your system considerably more difficult.

There can also be a concern with how the library or framework affects various aspects of your applications, notably security and privacy. If you use an external database, for example, you need to ensure that the data stored there meets the security and privacy constraints of your application. Similarly, if you build a text index over various users data as part of your application, you need to ensure you are maintaining each users privacy and security.

There is a risk that an external library might not be supported in the future and that its use by your application might become out-of-date. Our experience with JikesBT is one example of this. One should try to choose libraries that are actively maintained and have a substantial user base to avoid these problems.

External libraries and packages can also change over time. Fifteen years ago we built a package, $S^6$, that used code search to provide the developer with working code using code search **[SPR: ref]**, as seen in Figure 7.1. $S^6$ took as input a set of keywords, a signature for the method or class to generate, and one or more test cases. It used the keywords to do code search over a repository to get a set of possibly relevant files. It assumed each method in those files was a potential solution. It then took each solution and repeatedly applied transformations to it to get a new and possibly better solution. The transformations made the code compile; changed the signature to match that given as the input; removed unneeded code; added code from other parts of the file, for example fields and auxiliary methods; and generally made the code more likely to be correct. The resultant possible solutions, with the right signature and likely to compile, were then subjected to the test cases. Additional transforms were run after testing to fix minor problems such as string case errors and numerical off-by-one errors. The code that passed the test cases was then reformatted and passed back to the programmer. The whole process generally took under a minute.

Because we did not want to create and maintain our own code repository, we had $S^6$ use external code search packages, being careful to design the system so we could try out different packages. Over time, we have tried ten or more packages, with mixed success. Some of these external packages, such as Google Code Search, disappeared or were withdrawn. Others, such as GitHub code search, initiated rate limits on queries which severely affected $S^6$. Others only had a limited code base and had problems finding relevant code. All the code search engines had primitive search mechanisms, were very sensitive to the selection of keywords, and could not predictably return relevant results. As a result, $S^6$ had limited utility and was more of an experimental toy than a useful tool.

Intellectual property also needs to be considered in adapting any external library. Many companies are wary of using any external code because they fear that in doing so they might lose their trade secrets or might compromise the intellectual property rights of the code, possibly subjecting them to future lawsuits. Licences such as copyleft open source copyrights, which can be at the top level of a library or hidden deep in the required libraries, for example, can make your private code suddenly open source.

Next there is a cost to distribute your software it if requires outside packages, systems, or libraries. If you are only using the library on a server, this is not an issue. However, if code is being distributed to the user, you need to make sure you can distribute the libraries and then you have to ensure that a distribution of your code includes the necessary libraries and that these do not conflict with any system libraries that might be used.

For example, when we distribute the Code Bubble binary, we include our own copies of the various external libraries that are needed, for example junit, json, jtar, and jsyntaxpane. We also include a number of Eclipse libraries that we are able to distribute. Code Bubbles, however, requires an Eclipse installation. Our reading the the Eclipse license was that we are not able to provide an Eclipse installation ourselves, but rather have to go through the Eclipse Foundation to do so. This made our binary distribution and installation process much more complex.

In summary, using libraries, and open-source code in general, in your application is generally worthwhile and will save you time and effort both in the short term and as the system evolves and is maintained. However, care must be taken in choosing and incorporating these libraries to get their full benefits without adverse costs and risks.

## 7.2 Designing For Concurrency

Making the best use of today's hardware requires designing ones application to use concurrency. Today's CPUs have multiple cores and are used most efficiently when all the cores can be kept busy. Moreover, many applications have a server component that runs in the cloud, that is, on a cluster of machines with high-speed communications. These can make use of parallelism to provide higher throughput or to handle a large number of concurrent users.

Concurrency can also help improve applications and their design. Much of the run time of an application involves waiting, waiting for I/O to complete, waiting for a user to connect or reply, waiting for the memory to be paged in, or waiting for some other task to complete. Concurrency allows the application to make use of this idle time by using it for other computations. Concurrency can also simplify the design of the system by keeping independent things separate.

Many of the design decisions that involve concurrency are at the implementation or detailed design level. However, an understanding of concurrency and its possibilities and pitfalls can help create a stronger and more robust high-level design.

There are several ways that concurrency can arise in an application. The simplest is to have multiple independent processes that can run concurrently. The processes might communicate using messages or one might be executed by the other and generate results either in a pipe or a file. In either case, the first process should be able to do other work while the second is running. Multiple process concurrency is generally the easiest to design and code. The different processes are independent, can be written in different languages and by different teams. Each process is self-contained. The interaction between the processes is limited and generally well-defied. This limits, but by no means eliminates, the chance for concurrency problems such as deadlocks. One complication is that with multiple processes it is important for both processes to be able to detect the failure of the other and react accordingly.

Using multiple processes for concurrency is a well-used technique. The original C compiler used it to allow the preprocessor, the compiler, and the assembler to all run at once. We use it in Code Bubbles both to have Eclipse and the Code Bubbles front end do simultaneous work and to isolate separate components such as testing and version management from the Code Bubbles core.

A second instance of concurrency involves using multiple threads to handle background and input/output processing. This type of concurrency generally is inherent to detailed design and not that relevant to high-level design. Thread-based concurrency is useful when the tasks are too small to justify a separate process or where the tasks depend heavily on shared or common data. Background processing implies creating a task to generate a result that will be needed in the future. While it might be tempting to just create new threads for this purpose whenever there is a background task, a better design involves using common thread pools to simplify the overhead of creating and maintaining threads. Modern languages such as Dart and JavaScript provide direct support for this either using futures directly or using async and await keywords on methods.

Another use of multiple threads is to allocate a separate thread to handle I/O. This is especially useful for socket communication where one thread can handle attempts to connect to a server socket and then each connection gets its own thread to handle reading and writing on the actual connected socket. Using separate threads for I/O simplifies the code and the resultant logic. Separate threads work well as long as the number of connections remains relatively small. With hundreds or thousands of connections, alternative, more complex, approaches involving non-blocking I/O are required.

Another instance of concurrency inherent to modern applications involves the user interface. User interfaces in most modern systems run in their own thread. User interactions trigger callbacks to user routines from that thread. If the application needs to do some work in response to the user action, care must be taken. If the reaction can take significant time, generally over a fraction of a second, then it should be done in a separate thread or user interaction controlled by the user interface thread will be delayed. On the other hand, if the application wants to change the user interface based on the result of some computation, this can generally only be safely done within the user interface thread. Again, most of these concerns are involve detailed design rather than high-level design.

Another instance of concurrency involves concurrent algorithms and data structures. A lot of work has been done in developing parallel algorithms for specific problems and on developing data structures that are safe to use concurrently. Again, these are implementation concerns, and need not be considered during high-level design.

In general, high-level design should understand concurrency and how it might be used effectively in the application, but should otherwise ignore it. A high-level design can be thought of as synchronous, even when the architecture involves multiple processes. It is easier to reason about and understand a synchronous design and ensure that the overall design should work for the application.

At the same time, one should prepare for using concurrency later on in the development process. Various methods or functions in the interface can be identified as candidates for running concurrently or in background. If multiple users are to be supported, the design should ensure the users are handled independently and can be handled concurrently.

Any data structures that might need to be shared between threads should be identified as possibly concurrent. These should be isolated in a single component so that sharing is simpler and deadlocks and race conditions can be avoided. This might mean expanding the high-level design to create a new component, or it might mean adding to the interface of the component that holds the data structure new methods so that other components can access the shared data indirectly.

**[SPR: Do we need a prototyping section here or is it covered enough in previous chapter.]**

## 7.3 Summary

Summarize the chapter here.

Should state something about pros and cons of using external packages, emphasis on using well-understood and well-documented packages.

## 7.4 Further Reading

MySQL, PostgreSQL, Mongo manuals. OpenCV documentation, ...

## 7.5 Exercises

Modify program. Check if program can be expanded in various ways.

Identify risks and detail how they will be dealt with in your project design.

Identify what external code your project might or should use.

# 8. Design and the User Interface

User interfaces are an essential part of most modern applications. They are what users see. A bad user interface, one that is hard to use or unappealing to look at, can turn users off and cause the application to be a failure. A really good user interface can attract users even if the application's functionality is limited. Thus it is essential to design an application to accommodate its user interface.

Human-computer interaction (HCI), the study of how humans interact with computers and hence the study of what makes a good user interface and of how user interfaces should work, is a field of study unto itself. HCI can tell designers what user interface ideas work, what people like, and why they like them. HCI can tell designers how people prefer to interact with the computer. HCI can tell designers how to evaluate and assess user interfaces so that they can decide between multiple alternatives.

As a separate field, HCI requires a separate set of talents. Those who are strong programmers have one set of skills. Those who can create friendly, easy-to-use, and well liked user interfaces have a separate set. It is rare to find one person who has both sets of skills. HCI requires a creative, art-based background. Programming requires a more logical, mathematical background. Computer scientists typically design poor user interfaces.

Still, understanding the requirements for a user interface and how the user interface might fit into the application, even without actually designing the interface itself, is necessary for high level design.

## 8.1 Types of User Interfaces

*This page is intentionally left blank.*

# Bibliography