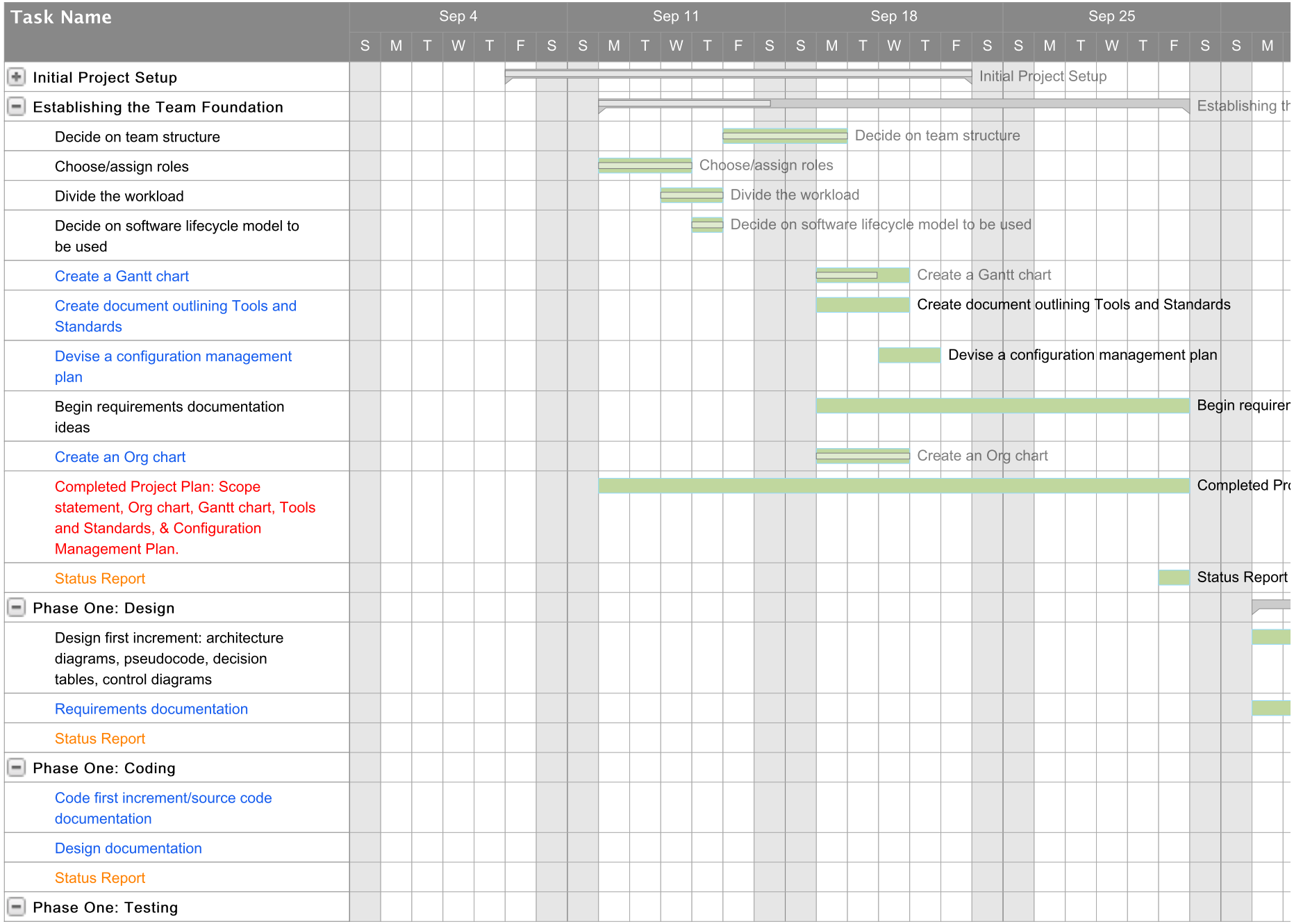
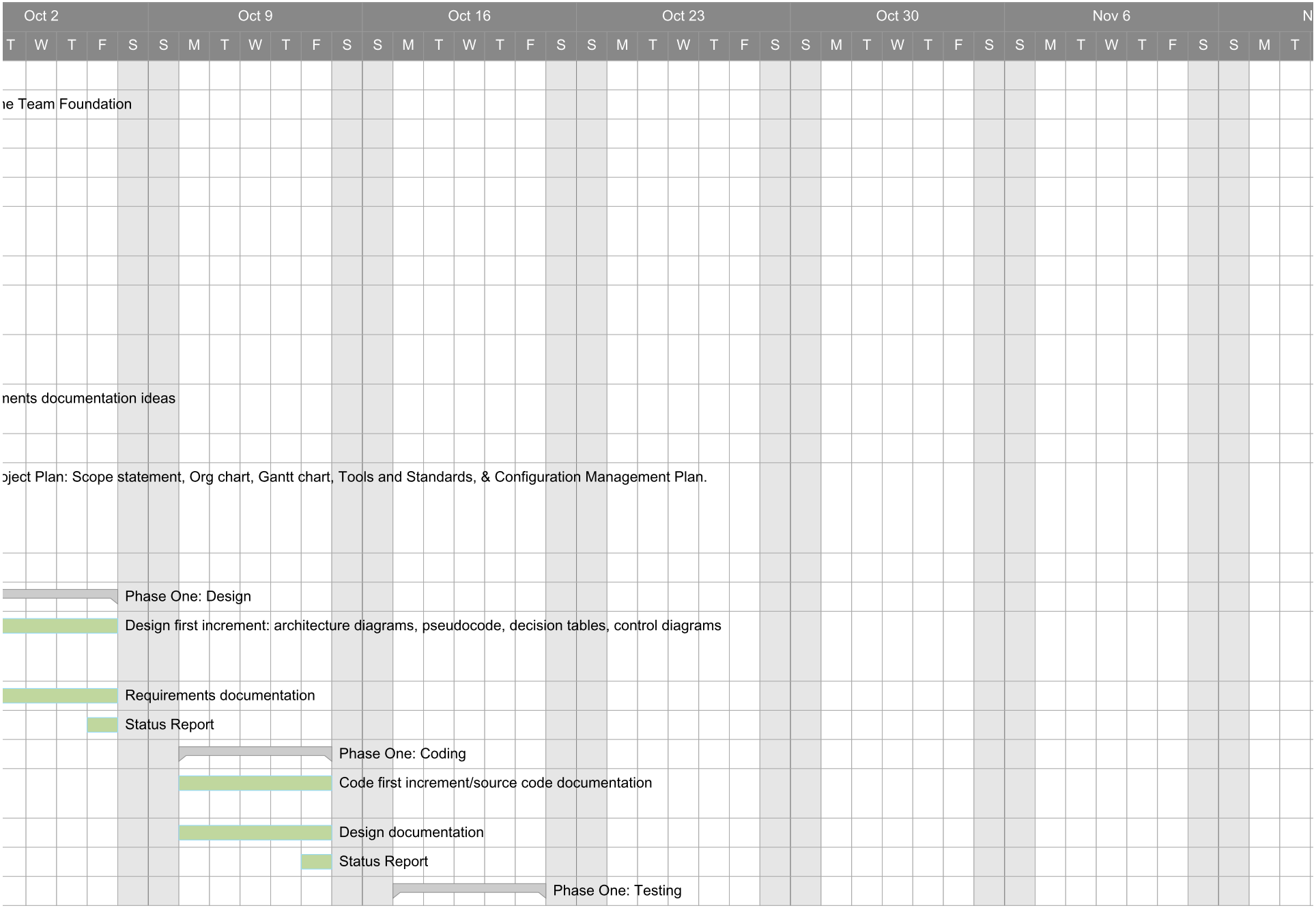
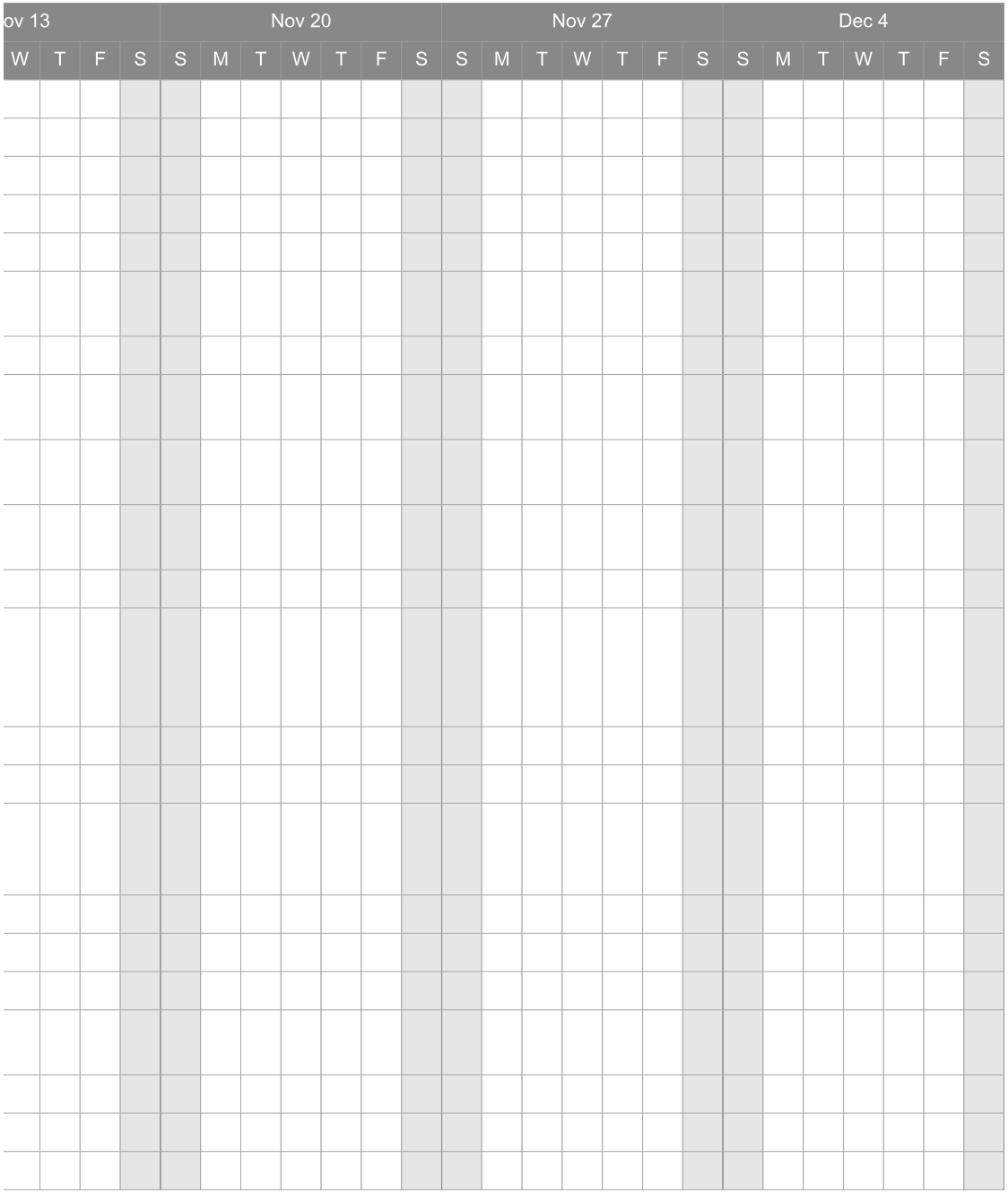
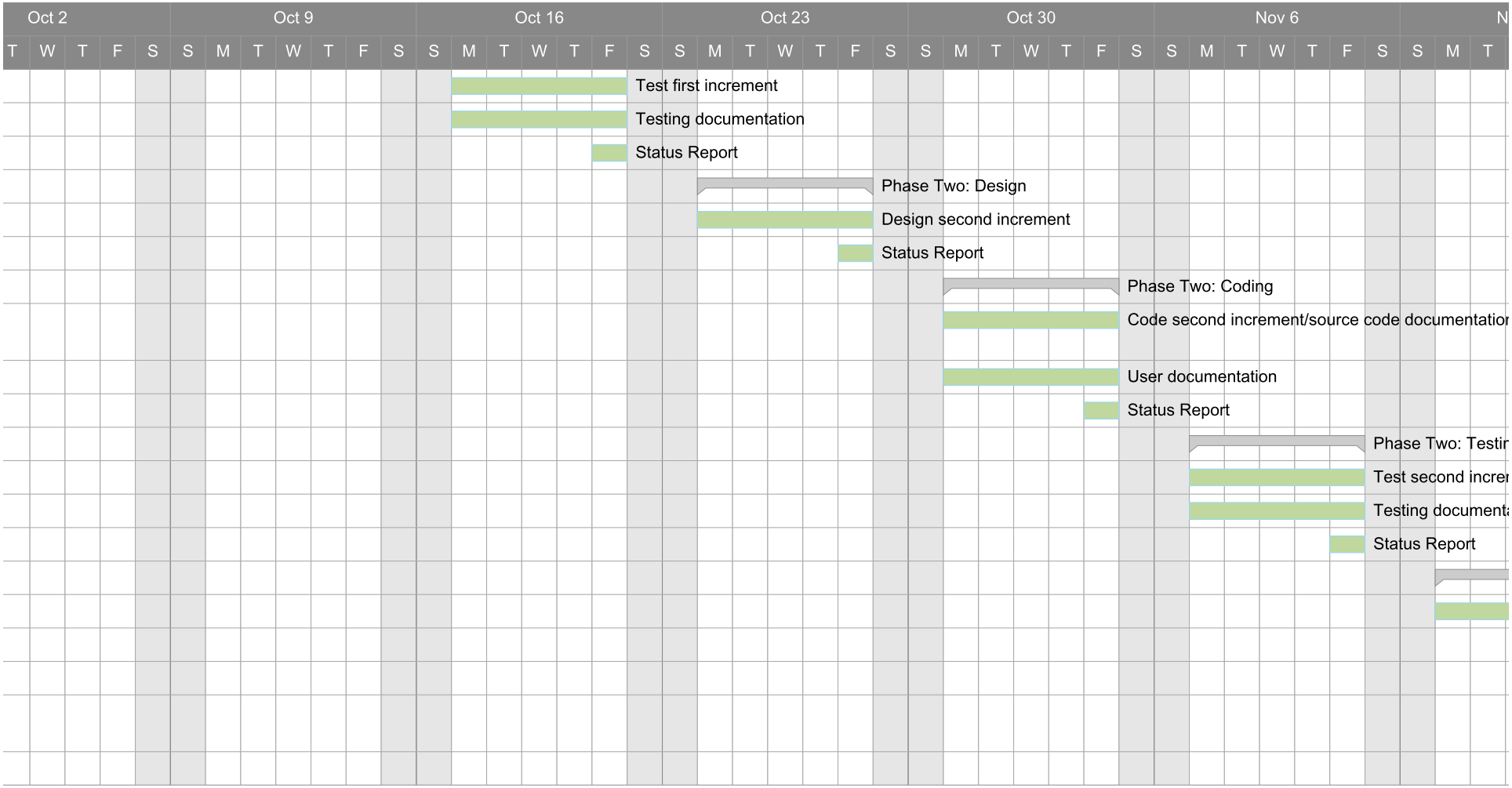
Team Primrose

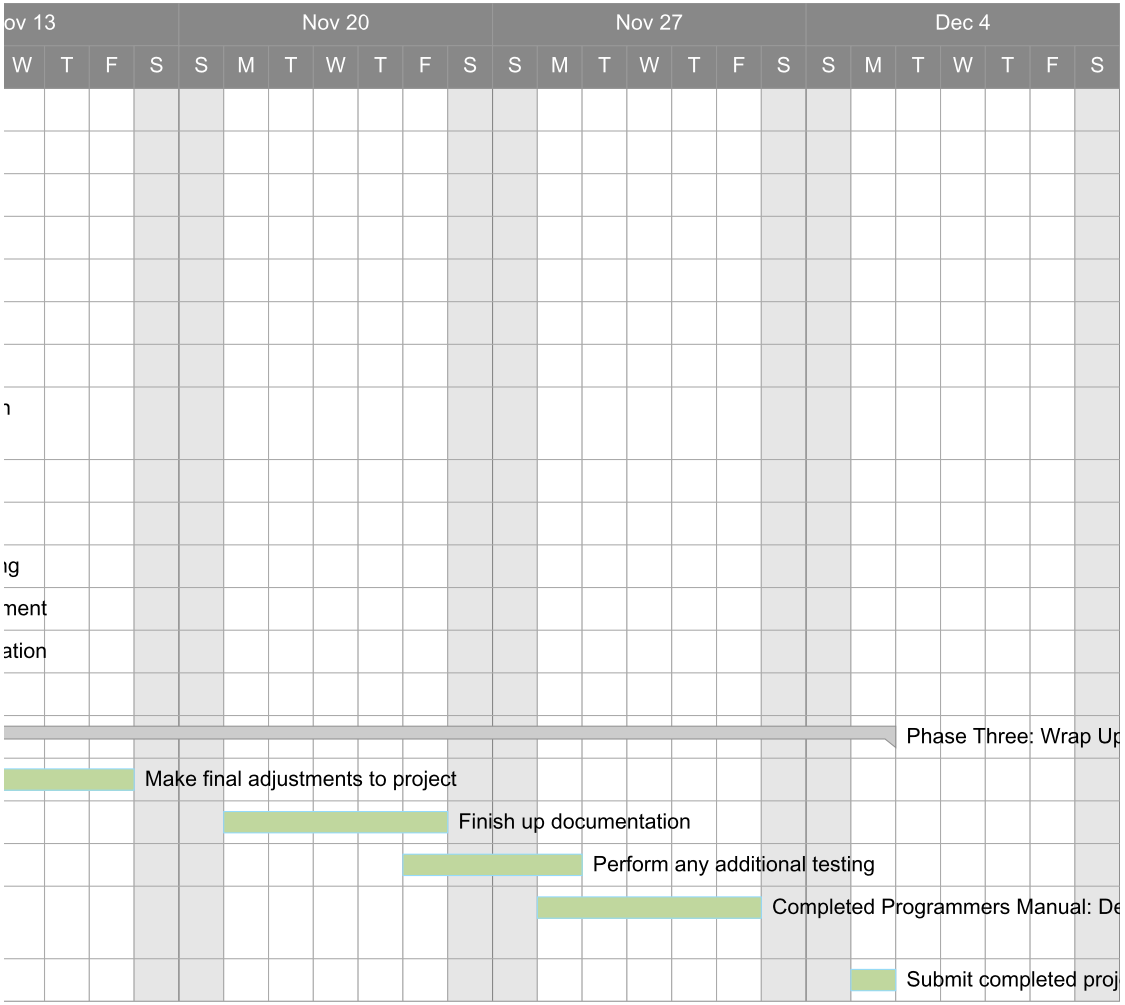



Oct 2
Oct 9
Oct 16
Oct 23
Oct 30
Nov 6
T W T F S S M T W T F S S M T W T F S S M T W T F S S M T W T F S S M T W T F S S M T W T F S S M T
ıe Team Foundation
ıents documentation ideas
ɔject Plan: Scope statement, Org chart, Gantt chart, Tools and Standards, & Configuration Management Plan.
Phase One: Design
Design first increment: architecture diagrams, pseudocode, decision tables, control diagrams
Requirements documentation
Status Report
Phase One: Coding
Code first increment/source code documentation
Design documentation
Status Report
Phase One: Testing


ov 13
Nov 20
Nov 27
Dec 4
W T F S S M T W T F S S M T W T F S S M T W T F S

	Task Name	Sep 4							Sep 11							Sep 18							Sep 25								
		S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M
29	Test first increment																														
30	Testing documentation																														
31	Status Report																														
32	**Phase Two: Design**																														
33	Design second increment																														
34	Status Report																														
35	**Phase Two: Coding**																														
36	Code second increment/source code documentation																														
37	User documentation																														
38	Status Report																														
39	**Phase Two: Testing**																														
40	Test second increment																														
41	Testing documentation																														
42	Status Report																														
43	**Phase Three: Wrap Up**																														
44	Make final adjustments to project																														
45	Finish up documentation																														
46	Perform any additional testing																														
47	Completed Programmers Manual: Design, Testing & User																														
48	Submit completed project																														


Oct 2
Oct 9
Oct 16
Oct 23
Oct 30
Nov 6
T W T F S S M T W T F S S M T W T F S S M T W T F S S M T W T F S S M T W T F S S M T W T F S S M T W T F S S M T
Test first increment
Testing documentation
Status Report
Phase Two: Design
Design second increment
Status Report
Phase Two: Coding
Code second increment/source code documentatio
User documentation
Status Report
Phase Two: Testi
Test second incre
Testing documenta
Status Report


ov 13
Nov 20
Nov 27
Dec 4
W T F S S M T W T F S S M T W T F S S M T W T F S
ıg
ment
ation
Phase Three: Wrap Up
Make final adjustments to project
Finish up documentation
Perform any additional testing
Completed Programmers Manual: De
Submit completed proj

Comments on Team Primrose

Row 1

Sample Discussion

Need help? E-mail us at support@smartsheet.com or call us at 425.283.1870

Todd Jones on 08/28/13 1:58 PM

> This is a sample discussion. Discussions will let you track comments about a particular line item or sheet. Many customers paste relevant emails into a discussion as well.
>
> *Todd Jones on 08/28/13 1:59 PM*

Row 6

Before going full speed ahead on gathering requirements, you should establish the boundaries of the software you will build. These boundaries should be documented in a scope statement. Examples of things that establish software boundaries are:

• What type of platform (Windows, Macintosh, UNIX, etc.) must the software work with?
• Will the software function as a standalone application on a given computer, or will it function over a network connection?
• What other software, if any, must the software interact with? For example, you might be building a subsystem component that will be integrated into a larger system. In such a case, it's important that you don't duplicate functionality provided by existing subsystems.
• What programming language will be used for the project?
• Will the software use a graphical interface or a command line interface?

Note that these are also part of the software requirements, and should therefore be included in your requirements document.

Matt Rhodes on 09/28/16 6:18 PM

Row 13

Indicate all the tasks you will perform, and how long you estimate you will spend on each task. You can also indicate on the chart which team members will be working on which tasks.

Matt Rhodes on 09/28/16 6:20 PM

Row 14

You should have a document that lists the various tools you used to build the project. This list should include things like programming language(s), editors, IDEs (Integrated Development Environments, e.g. Visual Studio, Eclipse), design tools such as UML modeling tools, etc.

For standards, you should include things like the process model you plan to follow, along with any standards (e.g., coding standards, documentation standards, etc.) you plan to use.

Matt Rhodes on 09/28/16 6:21 PM

Row 15

You should have a document that explains how you plan to maintain and keep track of the various versions you will end up developing. Be sure you clearly distinguish between your development versions and your release versions.

You need to have some kind of configuration management plan, since you will undoubtedly generate many versions of your software over the course of the semester. I suggest using something simple, like making a copy of the entire project folder for each new version, and using a numbering system like the standard major.minor.build or something similar. If you happen to run into storage problems while doing this, you can always prune the oldest versions from your hard drive and archive those to CD-ROM.

Whatever you do, you should always take care to make sure you keep a copy of versions that work. Many a project has been ruined by making one simple change that ended up cascading into a series of subsequent simple changes, to the point where the software became unstable, and the developer was unable to get back to the original, working version.

Matt Rhodes on 09/28/16 6:22 PM

Row 17

Include a simple org chart showing how your team is organized, and what each team member's role is.

Matt Rhodes on 09/28/16 6:19 PM

Row 22

Requirements Documentation

Detailed requirements are essential for building valid software (i.e., the software the customer wants). Your requirements should be as complete and as detailed as possible. You should organize your requirements in a sensible, hierarchical manner. Each requirement should be given a unique identifier that can be used to trace that requirement through the subsequent phases of the software life cycle. Your requirements document should follow the standard below, which is based on the IEEE/ANSI 830-1998 standard. For your requirements document, you do not need to include appendices (unless you have need to) or an index (an index would be nice, but unless you have a tool to generate an index automati-
cally, this is too time-consuming).

1. Introduction

1.1 Purpose of the requirements document - ignore this one

1.2 Scope of the product - just use your scope statement from your project plan here

1.3 Definitions, acronyms, and abbreviations - list and define all of these that appear in your documentation

1.4 References - include references for anything that is trademarked or copyrighted
here (e.g., game instructions, algorithms such as the LZW compression algorithm for images etc.)

1.5 Overview of the remainder of this document - ignore this one

2. General description

2.1 Product perspective - give a little background about why the software is being built and why it will be useful; one good paragraph will do

2.2 Product functions - describe fundamentally what the software will do; again, one good paragraph will suffice, and you do not need to go in depth (the specific requirements section will cover the in-depth functions)

2.3 User characteristics - describe who the end users will be

2.4 General constraints - describe the general limitations of your software (e.g., sys-

tem limitations, does not work over a network, etc.).5 Assumptions and dependencies - examples of these would be: assumes end user has QuickTime installed; requires Visual Basic runtime to be installed on end user's computer

3. Specific requirements

Section 3, which deals with the specific requirements, will vary depending on the software being built. I suggest using a simple numbering scheme. Below I show a partial set of requirements for a poker game, using a similar system. These requirements are far from complete - they are simply provided as an example.

Playing Cards

1.0.0 The game should use a standard deck of 52 playing cards, with no jokers.

1.1.0 User should be allowed to choose the design used for the card backs from 2 provided designs.

1.1.1 Solid blue design should be provided.

1.1.2 Red thatched design should be provided.

1.2.0 Each non-face card should display the appropriate numerical value

1.3.0 Each non-face card should display the appropriate number of suit symbols.

Game types

2.0.0 User should be allowed to choose the type of poker game they wish to play.

2.1.0 The game should allow the user to play 5-card draw, according to standard rules.

2.2.0 The game should allow the user to play 7-card stud, according to standard rules.

2.3.0 The game should allow the user to play Texas Hold'em, according to standard rules.

4. Appendices

You don't need to include appendices unless you feel they are warranted.

5. Index

Do not bother including an index unless you have a tool that can automatically

generate an index for a document. Otherwise this will be too time-consuming.

Matt Rhodes on 09/28/16 6:32 PM

Row 26

Design Documentation

Numerous design techniques exist, several of which are described in Sommerville's book. Which techniques you should use for a given project depends on the type of project, and what works best for you and your team. Some techniques are not suitable for some programs. For example, a detailed data flow diagram will probably be of little use if you are building a simple calculator application. The overriding goal is that the designs should be understandable. They should clearly indicate how the requirements will be implemented. Some of the more useful, and versatile, techniques are:

1. Architecture diagrams

— These show the various components that will be used and how they fit together. For software built using an object-oriented language, a UML class diagram is essential. If a procedural language is used, a diagram showing the hierarchical relationship of the various modules and methods works well. You

must have at least one architecture diagram in your design documentation - a diagram that shows how the various components of your software fit together.

2. Pseudocode

— Pseudocode is extremely useful when designing individual methods. The use of pseudocode has all but replaced the common flow chart. If done well, it can be incorporated into comments used for source code.

3. Decision Tables

— Decision tables are valuable when a course of action depends on a particular combination of values. A decision table shows all possible combinations of a set of values, and indicates what should happen for each combination.

4. Control Diagrams

— These include such diagrams as state-transition diagrams and activity diagrams. You should indicate how your requirements trace to your designs, by identifying each design component using the identifier(s) of the requirement(s) that component is designed to meet. The following hypothetical pseudocode shows an example:

```
// Pseudocode for allowing user to choose variant of poker to play
// (Requirement 2.0.0)
        //
// Assumes user has chosen the “New Game” menu option
display GUI dialog showing available game variants (5-card draw, 7-card stud,
        Texas      Hold’em)
if (choice == 5-card draw)
initialize 5-card draw game
else if (choice == 7-card stud)
initialize 7-card stud game
else if (choice == Texas Hold’em)
initialize Texas Hold’em game
```

One or two simple diagrams won’t cut it for your design documentation. Your project should be complex enough to require more than that. But there is no “target” number of diagrams necessary to get all the points for this part. I can’t set a target because I don’t know in advance what your project will be, and even if I did, I have no way of knowing what approach you will take to build the project. Keep in mind, the goal is for you to understand the problem you are solving before you start writing copious amounts of code. You need to convince me that you didn’t just sit down at your source code editor and hack out the program with all the designs in your head, or made up as you went along.

Matt Rhodes on 09/28/16 6:36 PM

Row 30

You are free to use whatever code documentation standard you are most comfortable with. The code writers on your team should try to use the same documentation standard, if possible. The only stipulations I have are:
1. Your source code must be documented to receive full points for that part of the project.
2. You should indicate the traceability from your designs to your code, by including the identifier of each design component (which should correspond to the identifier of the appropriate requirement) in the header comments of the source code used to implement that component. For example:

```
/**
* Query user for background pattern to use for cards
* (Requirement 1.1.0)
                */
        public    int    getCardBackground();
```

Matt Rhodes on 09/28/16 6:39 PM

Row 37

User Documentation

The User Documentation includes all documentation an end user would need to install and use the software.

1. User's Manual

You should assume that your software will be used by someone who has only a general idea of what the software is supposed to do. For example, if you're building a word processor, a user may know that your software is supposed to allow them to create letters and other such types of documents, but they won't necessarily know things like changing fonts, numbering pages, or that they can copy and paste sections of text using CTRL-C and CTRL-V.

1.1 Your user's manual should be written so that a user like this should be able to use your software effectively AFTER they have read the user's manual. Don't make any assumptions about the intuitive capabilities of the end users.

1.2 Although it does require more effort, supplementing the text in your user's manual with screenshots is highly recommended. Most companies these days no longer put the user manuals for their software in a separate document. Instead they embed the user manual in the software itself, and the user can usually access it from a "Help" menu option. You may integrate your user's manual into the software if you wish, or you can have a separate document.

2. Installation Instructions

Whether you include an installer for your software or not, you should include a document that instructs the user how to install your software and launch it. This document can be part of your user manual, or it can be a separate document. Either way, the instructions for installing your software should be prominently located in your user documentation. You should include the following information (at a minimum):

2.1 System requirements

2.2 Installation instructions

2.3 Any known issues that might affect installation or use

2.4 Basic troubleshooting if the software won’t install

Matt Rhodes on 09/28/16 6:49 PM