UCLA COMPUTER SCIENCE FIELD NOTES

CS 130

Software Engineering

Specification, modular design, architecture, verification, testing, teamwork, delivery, and evolution

How these notes are written. I wrote each chapter the way I wish the material had been explained the first time: the real idea, the point where I usually get stuck, the simplest mental model that still stays honest, and a worked decision instead of a polished answer appearing from nowhere. Every chapter closes with practice and a fully written solution. If a sentence sounds impressive but does not help me solve or explain something, it does not belong here.

Daniel Mastick

Curriculum map, working notes, practice, and solutions

Course map

Week	Core thread	What should be solid by the end
1	**Requirements, specifications, and observable contracts**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
2	**Modularity, abstraction, and abstract data types**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
3	**Structured design and software architecture**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
4	**Program reasoning and verification**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
5	**Testing strategy and defect isolation**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
6	**Version control, review, and team programming**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
7	**Builds, continuous integration, and release control**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
8	**Maintenance, refactoring, and socio-technical quality**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
9	**Integration workshop**	Connect at least three chapters in one end-to-end problem and explain where their contracts meet.
10	**Synthesis and project review**	Audit assumptions and explain a complete solution from interface to failure handling.

Scope anchor: https://catalog.registrar.ucla.edu/course/2025/COMSCI130. The sequence is aligned to the official catalog description and expanded into a practical ten-week study plan.

How I would use this packet

Treat each numbered section as one chapter. Read the formal explanation, pause at the student interpretation, and see whether the easy version still says the same thing. Rework the example without looking, then cover the solution in the chapter practice box. The cumulative set at the end is deliberately mixed: knowledge becomes durable when I have to choose the tool instead of being told which chapter I am in.

Contents

CHAPTER 1 Requirements, specifications, and observable contracts

The idea

Software engineering separates stakeholder goals from testable requirements and implementation choices. Preconditions, postconditions, invariants, acceptance criteria, nonfunctional constraints, and traceability make behavior inspectable.

Rebuild the chapter from first principles

The claim I need to own is this: Software engineering separates stakeholder goals from testable requirements and implementation choices. Preconditions, postconditions, invariants, acceptance criteria, nonfunctional constraints, and traceability make behavior inspectable. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of requirements, specifications, and observable contracts. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach requirements, specifications, and observable contracts on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from the course foundations to modularity, abstraction, and abstract data types. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Software engineering separates stakeholder goals from testable requirements and implementation choices.
- **Mechanism.** Preconditions, postconditions, invariants, acceptance criteria, nonfunctional constraints, and traceability make behavior inspectable.
- **Boundary.** The useful test is whether I can apply requirements, specifications, and observable contracts to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when requirements, specifications, and observable contracts is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

If I cannot write how a user or another module can tell the requirement was met, the requirement is not finished.

The easy version

Software engineering separates stakeholder goals from testable requirements and implementation choices.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Rewrite 'the search must be fast' as a

usable requirement. The conclusion is Define workload, dataset, service tier, percentile, threshold, and measurement method, such as: for a 10-million-record corpus, 95 percent of searches return within 250 ms under 200 concurrent users.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Rewrite 'the search must be fast' as a usable requirement.
Solution. Define workload, dataset, service tier, percentile, threshold, and measurement method, such as: for a 10-million-record corpus, 95 percent of searches return within 250 ms under 200 concurrent users.

Practice 2. Give a short oral explanation of requirements, specifications, and observable contracts that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct requirements, specifications, and observable contracts from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a requirements, specifications, and observable contracts problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 2 Modularity, abstraction, and abstract data types

The idea

Modules hide decisions behind cohesive interfaces. Abstract data types define legal operations independently of representation; information hiding, coupling, cohesion, dependency direction, and representation invariants govern change cost.

Rebuild the chapter from first principles

The claim I need to own is this: Modules hide decisions behind cohesive interfaces. Abstract data types define legal operations independently of representation; information hiding, coupling, cohesion, dependency direction, and representation invariants govern change cost. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of modularity, abstraction, and abstract data types. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach modularity, abstraction, and abstract data types on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from requirements, specifications, and observable contracts to structured design and software architecture. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Modules hide decisions behind cohesive interfaces.
- **Mechanism.** Abstract data types define legal operations independently of representation; information hiding, coupling, cohesion, dependency direction, and representation invariants govern change cost.
- **Boundary.** The useful test is whether I can apply modularity, abstraction, and abstract data types to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when modularity, abstraction, and abstract data types is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I judge a boundary by what future change it contains, not by how many files it creates.

The easy version

Modules hide decisions behind cohesive interfaces.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Why should clients not depend on a stack's internal array? The conclusion is Representation exposure lets clients violate invariants and makes resizing or replacing the array a breaking change. Clients should depend only on push, pop, top,

and the stated contract.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Why should clients not depend on a stack's internal array?
Solution. Representation exposure lets clients violate invariants and makes resizing or replacing the array a breaking change. Clients should depend only on push, pop, top, and the stated contract.

Practice 2. Give a short oral explanation of modularity, abstraction, and abstract data types that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct modularity, abstraction, and abstract data types from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a modularity, abstraction, and abstract data types problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 3 Structured design and software architecture

The idea

Architecture assigns responsibilities and communication across layers, services, events, repositories, and adapters. Decomposition should expose data ownership, failure boundaries, security boundaries, latency, and deployment consequences.

Rebuild the chapter from first principles

The claim I need to own is this: Architecture assigns responsibilities and communication across layers, services, events, repositories, and adapters. Decomposition should expose data ownership, failure boundaries, security boundaries, latency, and deployment consequences. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of structured design and software architecture. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach structured design and software architecture on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from modularity, abstraction, and abstract data types to program reasoning and verification. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Architecture assigns responsibilities and communication across layers, services, events, repositories, and adapters.
- **Mechanism.** Decomposition should expose data ownership, failure boundaries, security boundaries, latency, and deployment consequences.
- **Boundary.** The useful test is whether I can apply structured design and software architecture to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when structured design and software architecture is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

A box-and-arrow diagram becomes useful only when I can explain what crosses each arrow and what happens when it fails.

The easy version

Architecture assigns responsibilities and communication across layers, services, events, repositories, and adapters.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: When is an event queue preferable to a synchronous call? The conclusion is Use it when producers should not wait for consumers, bursts need buffering, or work can be retried asynchronously. The design must accept eventual consistency, duplicate delivery, ordering limits, and observability needs.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. When is an event queue preferable to a synchronous call?
Solution. Use it when producers should not wait for consumers, bursts need buffering, or work can be retried asynchronously. The design must accept eventual consistency, duplicate delivery,

ordering limits, and observability needs.

Practice 2. Give a short oral explanation of structured design and software architecture that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct structured design and software architecture from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a structured design and software architecture problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 4 Program reasoning and verification

The idea

Assertions, invariants, weakest preconditions, type systems, static analysis, model checking, and proofs provide evidence beyond sampled tests. Verification claims are always relative to a specification, environment, and model.

Rebuild the chapter from first principles

The claim I need to own is this: Assertions, invariants, weakest preconditions, type systems, static analysis, model checking, and proofs provide evidence beyond sampled tests. Verification claims are always relative to a specification, environment, and model. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of program reasoning and verification. List the known quantities, the unknown, the units or types, and any hidden constraint.

2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach program reasoning and verification on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from structured design and software architecture to testing strategy and defect isolation. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Assertions, invariants, weakest preconditions, type systems, static analysis, model checking, and proofs provide evidence beyond sampled tests.
- **Mechanism.** Verification claims are always relative to a specification, environment, and model.
- **Boundary.** The useful test is whether I can apply program reasoning and verification to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when program reasoning and verification is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

A proved program can still solve the wrong problem, so I keep the specification beside the proof obligation.

The easy version

Assertions, invariants, weakest preconditions, type systems, static analysis, model checking, and proofs provide evidence beyond sampled tests.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Give a loop invariant for finding the maximum of a nonempty array. The conclusion is Before processing index i, best equals the maximum of elements zero through i minus one. Initialization uses the first element, each comparison preserves the claim, and termination covers the array.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Give a loop invariant for finding the maximum of a nonempty array.
Solution. Before processing index i, best equals the maximum of elements zero through i minus one. Initialization uses the first element, each comparison preserves the claim, and termination covers the array.

Practice 2. Give a short oral explanation of program reasoning and verification that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct program reasoning and verification from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a program reasoning and verification problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 5 Testing strategy and defect isolation

The idea

Unit, integration, contract, system, property-based, fuzz, performance, security, and acceptance tests answer different questions. Good suites partition input space, target boundaries, control nondeterminism, and identify broken contracts.

Rebuild the chapter from first principles

The claim I need to own is this: Unit, integration, contract, system, property-based, fuzz, performance, security, and acceptance tests answer different questions. Good suites partition input space, target boundaries, control nondeterminism, and identify broken contracts. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of testing strategy and defect isolation. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach testing strategy and defect isolation on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from program reasoning and verification to version control, review, and team programming. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Unit, integration, contract, system, property-based, fuzz, performance, security, and acceptance tests answer different questions.
- **Mechanism.** Good suites partition input space, target boundaries, control nondeterminism, and identify broken contracts.
- **Boundary.** The useful test is whether I can apply testing strategy and defect isolation to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when testing strategy and defect isolation is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

Coverage tells me where tests executed, not whether they asked a meaningful question.

The easy version

Unit, integration, contract, system, property-based, fuzz, performance, security, and acceptance tests answer different questions.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: What property can test a sorting function beyond one expected array? The conclusion is For generated inputs, assert the output is ordered, contains exactly the same multiset, has the same length, and is idempotent under sorting again. Add oracle checks on small cases.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. What property can test a sorting function beyond one expected array?
Solution. For generated inputs, assert the output is ordered, contains exactly the same multiset, has the same length, and is idempotent under sorting again. Add oracle checks on small cases.

Practice 2. Give a short oral explanation of testing strategy and defect isolation that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct testing strategy and defect isolation from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a testing strategy and defect isolation problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 6 Version control, review, and team programming

The idea

Team development depends on small reviewable changes, coherent commits, branch policy, ownership, issue tracking, design records, code review, merge discipline, and conflict resolution.

Rebuild the chapter from first principles

The claim I need to own is this: Team development depends on small reviewable changes, coherent commits, branch policy, ownership, issue tracking, design records, code review, merge discipline, and conflict resolution. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of version control, review, and team programming. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach version control, review, and team programming on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from testing strategy and defect isolation to builds, continuous integration, and release control. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Team development depends on small reviewable changes, coherent commits, branch policy, ownership, issue tracking, design records, code review, merge discipline, and conflict resolution.
- **Mechanism.** The useful test is whether I can apply version control, review, and team programming to a new case and explain the assumption that makes the answer valid.
- **Boundary.** The useful test is whether I can apply version control, review, and team programming to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when version control, review, and team programming is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I make a commit tell one defensible story so a reviewer can reason about cause, risk, and rollback.

The easy version

Team development depends on small reviewable changes, coherent commits, branch policy, ownership, issue tracking, design records, code review, merge discipline, and conflict resolution.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Why separate a mechanical rename from a behavior change? The conclusion is The reviewer can verify each concern independently, diffs stay

legible, blame remains useful, and either change can be reverted without entangling the other.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Why separate a mechanical rename from a behavior change?
Solution. The reviewer can verify each concern independently, diffs stay legible, blame remains useful, and either change can be reverted without entangling the other.

Practice 2. Give a short oral explanation of version control, review, and team programming that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct version control, review, and team programming from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a version control, review, and team programming problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 7 Builds, continuous integration, and release control

The idea

Reproducible builds pin tools and dependencies; CI enforces fast feedback; staged delivery, feature flags, migrations, observability, rollback, and incident response control production risk.

Rebuild the chapter from first principles

The claim I need to own is this: Reproducible builds pin tools and dependencies; CI enforces fast feedback; staged delivery, feature flags, migrations, observability, rollback, and incident response control production risk. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of builds, continuous integration, and release control. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach builds, continuous integration, and release control on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from version control, review, and team programming to maintenance, refactoring, and socio-technical quality. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Reproducible builds pin tools and dependencies; CI enforces fast feedback; staged delivery, feature flags, migrations, observability, rollback, and incident response control production risk.
- **Mechanism.** The useful test is whether I can apply builds, continuous integration, and release control to a new case and explain the assumption that makes the answer valid.
- **Boundary.** The useful test is whether I can apply builds, continuous integration, and release control to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when builds, continuous integration, and release control is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

Shipping is a state transition with an escape route, not the moment the code leaves my laptop.

The easy version

Reproducible builds pin tools and dependencies; CI enforces fast feedback; staged delivery, feature flags, migrations, observability, rollback, and incident response control production risk.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: How should a breaking database migration be deployed safely? The conclusion is Expand the schema compatibly, deploy code that reads and writes both forms, backfill with monitoring, switch reads, then contract the old schema only after rollback and old clients are no longer needed.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. How should a breaking database migration be deployed safely?
Solution. Expand the schema compatibly, deploy code that reads and writes both forms, backfill with monitoring, switch reads, then contract the old schema only after rollback and old clients are no longer needed.

Practice 2. Give a short oral explanation of builds, continuous integration, and release control that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct builds, continuous integration, and release control from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a builds, continuous integration, and release control problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 8 Maintenance, refactoring, and socio-technical quality

The idea

Most software cost arrives after first release. Refactoring preserves observable behavior while improving structure; maintenance also includes debugging, dependency updates, security repair, performance work, documentation, deprecation, and product change.

Rebuild the chapter from first principles

The claim I need to own is this: Most software cost arrives after first release. Refactoring preserves observable behavior while improving structure; maintenance also includes debugging, dependency updates, security repair, performance work, documentation, deprecation, and product change. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of maintenance, refactoring, and socio-technical quality. List the known quantities, the unknown, the units or types, and any hidden

constraint.

2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach maintenance, refactoring, and socio-technical quality on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from builds, continuous integration, and release control to the cumulative course model. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Most software cost arrives after first release.
- **Mechanism.** Refactoring preserves observable behavior while improving structure; maintenance also includes debugging, dependency updates, security repair, performance work, documentation, deprecation, and product change.
- **Boundary.** The useful test is whether I can apply maintenance, refactoring, and socio-technical quality to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when maintenance, refactoring, and socio-technical quality is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

Technical debt is a future cost with interest, so I name the affected change and evidence instead of calling every ugly line debt.

The easy version

Most software cost arrives after first release.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: How is refactoring different from feature work? The conclusion is Refactoring changes internal structure while preserving the public behavioral contract. Feature work intentionally changes behavior; separating them improves tests, review, and diagnosis.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. How is refactoring different from feature work?
Solution. Refactoring changes internal structure while preserving the public behavioral contract. Feature work intentionally changes behavior; separating them improves tests, review, and diagnosis.

Practice 2. Give a short oral explanation of maintenance, refactoring, and socio-technical quality that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct maintenance, refactoring, and socio-technical quality from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a maintenance, refactoring, and socio-technical quality problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

Final study check

I should be able to close this packet and reconstruct the core models, not merely recognize their names. My final check is to explain one complete problem aloud: what the inputs mean, which invariant or contract controls the work, why the method is legal, what it costs, how it can fail, and what evidence would convince another engineer.