22CSC51 - AGILE METHODOLOGIES

Prepared By,

Mr.M.Muthuraja,

Assistant Professor

Department.of CSE

Kongu Engineering College

Unit - III

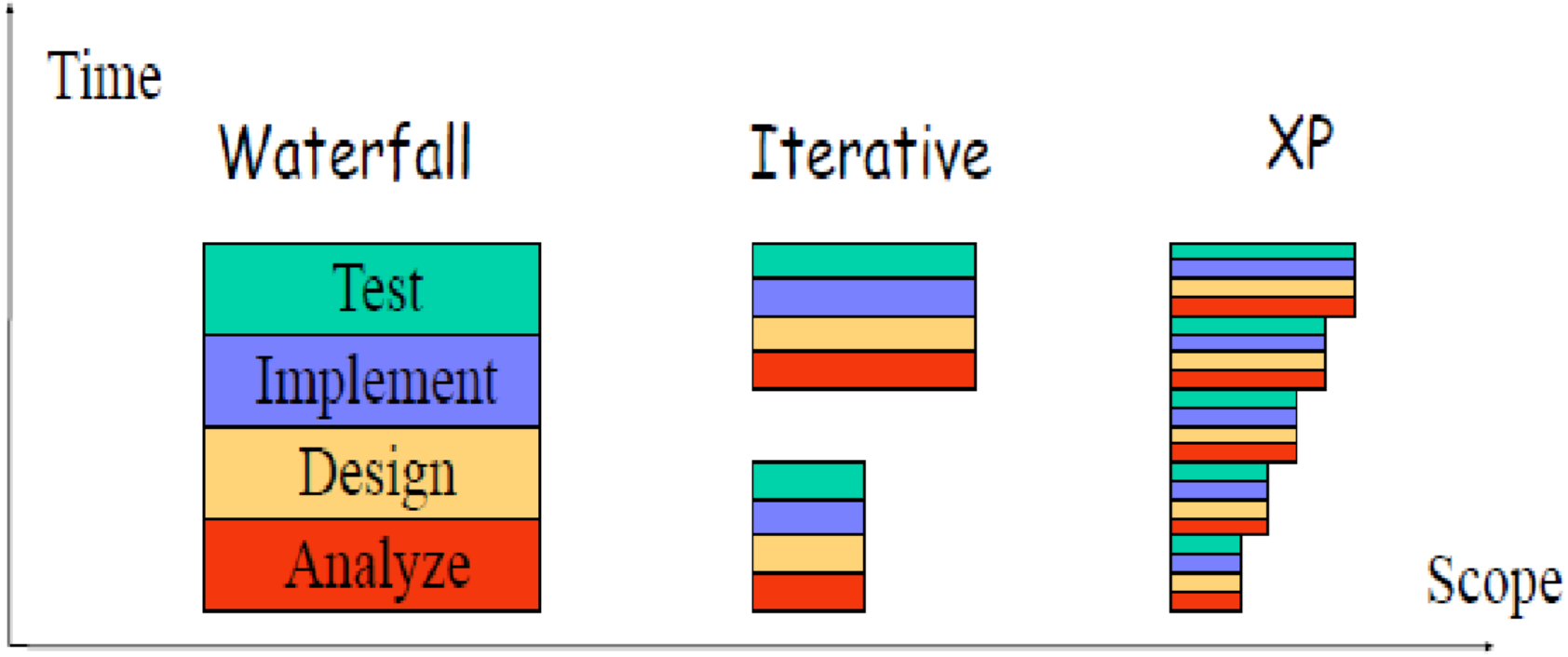
XP and Incremental Design, Lean, and Kanban

XP Extreme Programming

- XP is a **lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way** to develop **software**.
- Suitable for **small to medium sized** projects that works under vague and rapidly changing environment, usually ranged from **6 to 15 months**.
- XP was used by small teams ranging from **two to twelve members.**
- Kent Beck in 1999 invented XP to meet the changing requirements needed in the market place.
- **Goals:**
  - ✓ **Minimize unnecessary work**
  - ✓ **Maximize communication and feedback**
  - ✓ **Make sure that developers do most important work**
  - ✓ **Make system flexible, ready to meet any change in requirements**

Extreme Programming (XP)

- XP: like iterative but taken to the extreme



XP Variables

- XP regards a software development project as a system of four control "variables":
  1. **Cost**
  2. **Scope**
  3. **Quality**
  4. **Time**

Cost

The amount of money to be spent. The **resources** (how many developers, equipment etc.) available for the project are directly related to this variable.

Time

Determines when the **system (release) should** be done.

Quality

The **correctness of the system** (as defined by the customer) and how well tested it will be.

Scope

Describes what and **how much will be done** (functionality).

Relationship between Variables

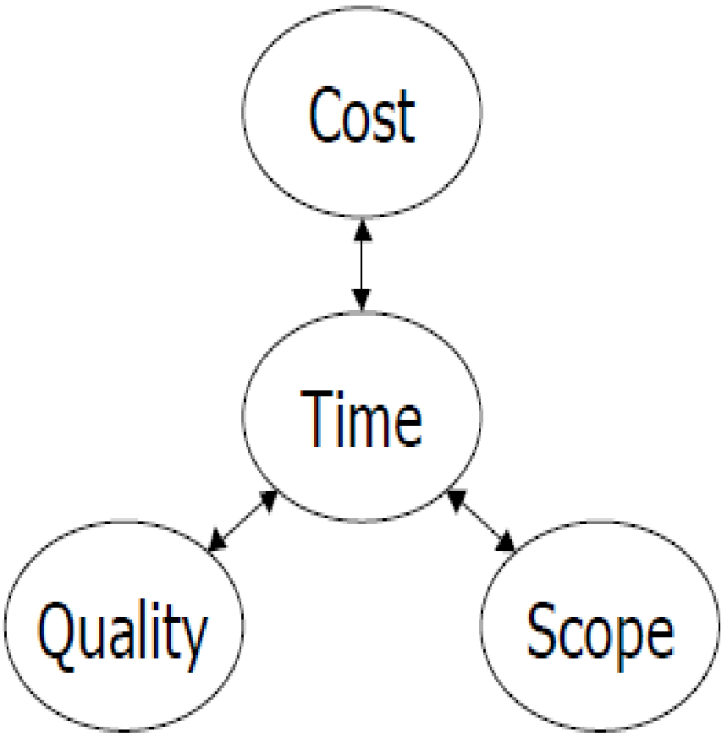


Figure 1 : The relationships between the variables

- Time is the **central variable** in XP.
- **Increasing quality** can increase the time
- **Increasing scope** means more time is needed
- **Increasing cost** (hiring more developers or providing better equipment) can mean less time

XP Values

XP defines five **"values"** which are used as guidelines throughout development.

- **Communication**
- **Simplicity**
- **Feedback**
- **Courage**
- **Respect**

XP Values

❖ Communication

- Good **communication is one of the key factors** necessary for the success of software projects.
- Customers **need to communicate** the requirements to the **developers**. Developers communicate ideas and design to each other and so on.
- A lot of problems can be **traced to a breakdown** in communication.
- XP tries to keep communication flowing in a variety of ways.

XP Values

❖ **Simplicity**

- XP strives for **simple systems**. This means, they should be **as simple as possible** but they **must work**.
- XP also strives for simplicity in the methodology.
- It **reduces the amount of artifacts** to an absolute minimum
  - **– the requirements (User Stories), plans (Planning Game) and the product (code).**
- The practices and techniques can be learned in a matter of hours
- The main reason for the desire for **simplicity** is that XP tries to cope with changes and other risks.
- Simplicity means that you always strive to **"Do The Simplest Thing That Could Possibly Work"**

XP Values

❖ Feedback

- XP is a **feedback-driven process**. You need feedback at all scales, whether you are a customer, manager or developer.
- In XP feedback is especially **relevant to business** because it is the base for influencing the process.
- Feedback has two important characteristics.
  - **The first is quality.**
  - **The second is time.**

XP Values

❖ Courage

- This is a **somewhat vague** value.
- It includes **courage** as well as a certain amount of **aggressiveness**.
- Courage is needed because a lot of the rules and practices are **"extreme"** in the way that they go against "tradition" or **"wisdom"** of software engineering.
- Aggressiveness is the **attitude towards the implementation** of the system.

XP Values

❖ **Respect**

- Team members **respect each other**
- When team members have **mutual respect**, they're able to **trust** each other to do a good job with the work they've taken on.
- But that respect doesn't always come easily to programmers and other technical people.
- A good **XP** finds ways to increase the team members' mutual respect for each other.

XP Principles

- Humanity
- Economics
- Mutual benefit
- Self similarity
- Improvement
- Diversity
- Reflection
- Flow
- Opportunity
- Redundancy
- Failure
- Quality
- Accepted responsibility
- Baby steps

XP Principles

❖ **Humanity**

- Remember that **software is built by people**, and there's a balance between the needs of each team member and the project's needs.

❖ **Economics**

- Somebody is always **paying for software project** and everybody has to keep the budget in mind.

❖ **Mutual benefit**

- Search for practices that **benefit the individual**, the **team,** and the customer together.

❖ **Self Similarity**

- The pattern of a **monthly cycle** is the same as a weekly cycle and the same as a daily cycle.

XP Principles

❖ **Improvement**

- Do your **best today,** and stay aware of what you need to do to get better tomorrow.

❖ **Diversity**

- Lots of different **opinions and perspectives** work together to come up with better solutions.

❖ **Reflection**

- Good teams stay aware of **what's working and what isn't in their software process.**

❖ **Flow**

- Constant delivery means a **continuous flow of development** work, not distinct phases.

❖ **Opportunity**

- Each problem the **team encounters is a chance to learn** something new about software development.

XP Principles

❖ **Redundancy**

- Even though it can seem **wasteful at first blus**h, redundancy avoids big quality problems.

❖ **Failure**

- You can **learn a lot from failing**. It's OK to try things that don't work.

❖ **Quality**

- You **can't deliver faster by accepting** a lower quality product.

❖ **Accepted responsibility**

- If someone is **responsible for something**, they need to have the authority to get it done.

❖ **Baby steps**

- Take **small steps in the right direction** rather than making wholesale changes when adopting new practices.

XP Practices

There are 13 Primary Practices . These are grouped into the following 4 Categories

- **Programming Practices**
- **Integration Practices**
- **Planning Practices**
- **Team Practices**

XP Practices

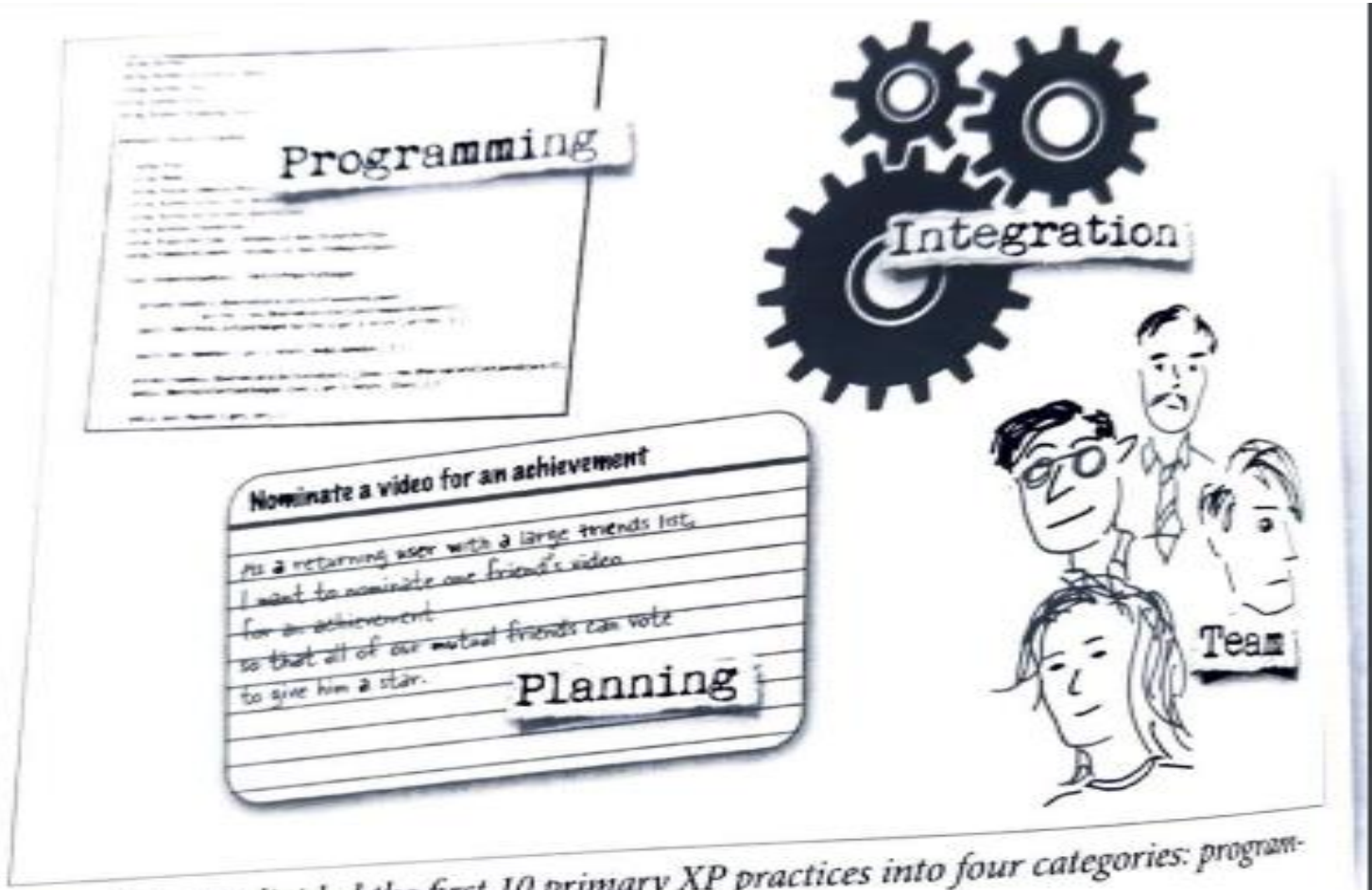


Figure 6-1. We divided the first 10 primary XP practices into four categories: programming, integration, planning, and team.

XP Practices

Business or Planning Practices

1. The Planning Game
2. Small Release
3. Metaphor
4. On Site Customer

Team Work Practices

1. Collective Ownership
2. Coding Standard
3. Refactoring
4. 40 Hours a week

Programming Practices

1. Simple Design
2. Test Driven Development
3. Pair Programming

Integration Practices

1. Continuous Integration
2. 10 Minute Build

XP Practices

- **The Planning Game—**Quickly determine the scope of the **next release by combining business priorities and technical estimates**. As reality overtakes the plan, update the plan.
- **Small releases—**Put a **simple system** into production quickly, then release new versions on a very short cycle.
- **Metaphor—**Guide all development with a simple shared story of how the **whole system works**.
- **Simple design—**The system should be designed as **simply as possible** at any given moment.
- Extra complexity is removed as soon as it is discovered.

XP Practices

- **Testing—**Programmers continually write **unit tests,** which must run flawlessly for development to continue.
- Customers write tests demonstrating that features are finished.
- **Refactoring—**Programmers **restructure** the system **without changing its behavior** to remove duplication, improve communication, simplify, or add flexibility.
- **Pair programming—**All production code is written with **two programmers at one** machine.
- **Collective ownership—**Anyone can **change any code** anywhere in the system at any time.

XP Practices

- **Continuous integration—Integrate and build** the system many times a day, every time a task is completed.
- **40 hour week—**Work **no more than 40 hours** a week as a rule.
- Never work overtime a second week in a row.
- **On-site customer—**Include a real, live user on the team, available full-time to answer questions.
- **Coding standards—**Programmers write all code in accordance with rules **emphasizing communication** through the code.

The Planning Game

- Neither **business** considerations nor **technical** considerations should be **paramount**.
- Software development is always an evolving dialog between the **possible and the desirable**.
- The nature of the dialog is that it changes both what is seen to be possible and what is seen to be desirable.
- **Business people** need to decide about
- **Scope—**How much of a **problem must be solved** for the system to be valuable in production?
- **Priority—**If you could only have A or B at first, which one do you want? The business person is in a position to determine this, much more so than a programmer.

The Planning Game

- **Composition of releases—**How **much or how little** needs to be done before the business is better off with the software than without it?
- **Dates of releases—**What are **important dates** at which the presence of the software (or some of the software) would make a big difference?
- **Technical people** decide about
- **Estimates—**How long will a feature take to implement?

The Planning Game

- **Consequences—**There are **strategic business** decisions that should be made only when informed about the technical consequences.
- **Process—**How will the **work and the team** be organized? The team needs to fit the culture in which it will operate.
- **Detailed scheduling—**Within a release, **which stories will be done first?** The programmers need the freedom to schedule the riskiest segments of development first, to reduce the overall risk of the project.

Small Releases

- Every **release should** be **as small as possible**, containing the most valuable business requirements.
- The release has to **make sense as a whole**—that is, you can't implement half a feature and ship it, just to make the release cycle shorter.
- It is **far better to plan a month** or **two at a time** than **six months or a year** at a time.
- A company shipping bulky software to customers **might not be able to release** this often.
- They should still reduce their cycle as much as possible.

Metaphor

- Extreme Programming teams develop a **common vision** of how the program works, which we call the **"metaphor"**.
- At its best, the **metaphor** is a **simple evocative description** of how the **program works**.
- Each XP software project is guided by a single overarching metaphor.
- The metaphor just **helps everyone on the project** understand the **basic elements and their relationships**.
- Metaphor is **something you start** using when your mother asks what you are working on and you try to explain her the details.
- How you find it is **very project-specific**. Use your common sense or find the guy on your team who is good at explaining technical things to customers in a way that is easy to understand.

Simple Design

- Always use the **simplest possible design** that gets the **job done**.
- The requirements will **change tomorrow,** so only do **what's needed** to meet **today's requirements**.
- The system should be designed **as simply as possible** at any given moment.
- Extra **complexity is removed** as soon as it is discovered.

Test-Driven Development (TDD)

- In XP **Developers** write the unit **test cases before starting the coding**.
- The team automates the unit tests and it helps during the build and integration stage.
- The main **benefit of TDD** is that programmers have to only **write the code** that passes the tests.

Basics steps of TDD,

- Write the **unit test case first**
- Write a **minimal amount** of code to pass the test.
- **Refactor** it by adding the needed feature and functionality, While continuously making sure the tests pass.

Refactoring

- XP uses **Refactoring to enhance** the design of a system such that it is **easier to understand** and **modify without** affecting the behavior.
- If a Refactoring would not enhance the system, then it is **not applied.**
- Furthermore, Refactoring is only done as part of the development cycle: **before and after the implementation of a task** (before implementation to ease the integration of the new code).
- Refactoring **do not affect the behavior**.
- Refactoring can have a **slightly negative effect** on **performance**, but the code tends to be better suited to optimization later on.

Pair Programming

- In a team that does **pair programming, two developers sit** together at a **single workstation** when writing code.
- In most cases, **one programmer types**, while the other **watches—**
  - **Is the overall system going to work**
  - **Are there better ways of doing this**
  - **What test cases still don't work**

  and they constantly discuss what's going on.
- Teams working in **pairs inject far fewer bugs**, because there are two sets of eyes looking to catch the bugs all the time.
- Pairs switch **roles frequently** (**every two hours**)

Collective Ownership

- The **code** (classes etc.) created in an XP project is owned by the **complete team**, not by the individual **developers**.
- This is important because **anyone has to be able** to modify anything if necessary.
- Any **developer is expected** to be able to work on any part of the codebase at any time.
- However problems with Collective Code Ownership arise when

✓ **a strong ownership mentality exists**

✓ **non-project components are (re)used**

✓ **expert knowledge is needed**

Continuous Integration

- All changes are **integrated into the codebase** at least daily.
- Unit tests have to **run 100%** both **before and after** integration.
- The whole system is **built and tested** several times a day
- The benefits of frequent integration are :

❖ Integration is **easier** because the changes are small.

❖ The unit test suite ran fine before integration.

❖ If **problems are discovered** by the suite they probably are located in the changes.

❖ The **codebase represents** the current state of the system. If a snapshot is needed it can be **created anytime** from the codebase stored in the integration computer.

10 – Minute Build

- The **team creates** an automated build for the entire code base that runs in under **10 mins**
- This build includes **automatically running** all of the unit tests and generating a report of which test passed and which failed
- If a build takes **more than 10 minutes** to run, team members are **much less likely to run** it often.

40 Hours a week

- XP projects **emphasize the forty hour week**. This means that the team should avoid to **work overtime**.
- Two reasons are given for this rule :
- "Projects that **need working overtime** will be **too late anyway.**"
- Usually either the **schedule is too tight** (business controls both time and scope) or the team couldn't cope in time with unexpected risks. The only way to deal with the latter is to **re-negotiate the schedule,** otherwise the quality will be decreased.
- · The motivation will be affected. For most of XP's rules and practices it is necessary that the team members are rested and motivated.
- In practice, **XP projects rarely require** working **overtime**, and if so, at most one or two times during the whole project.

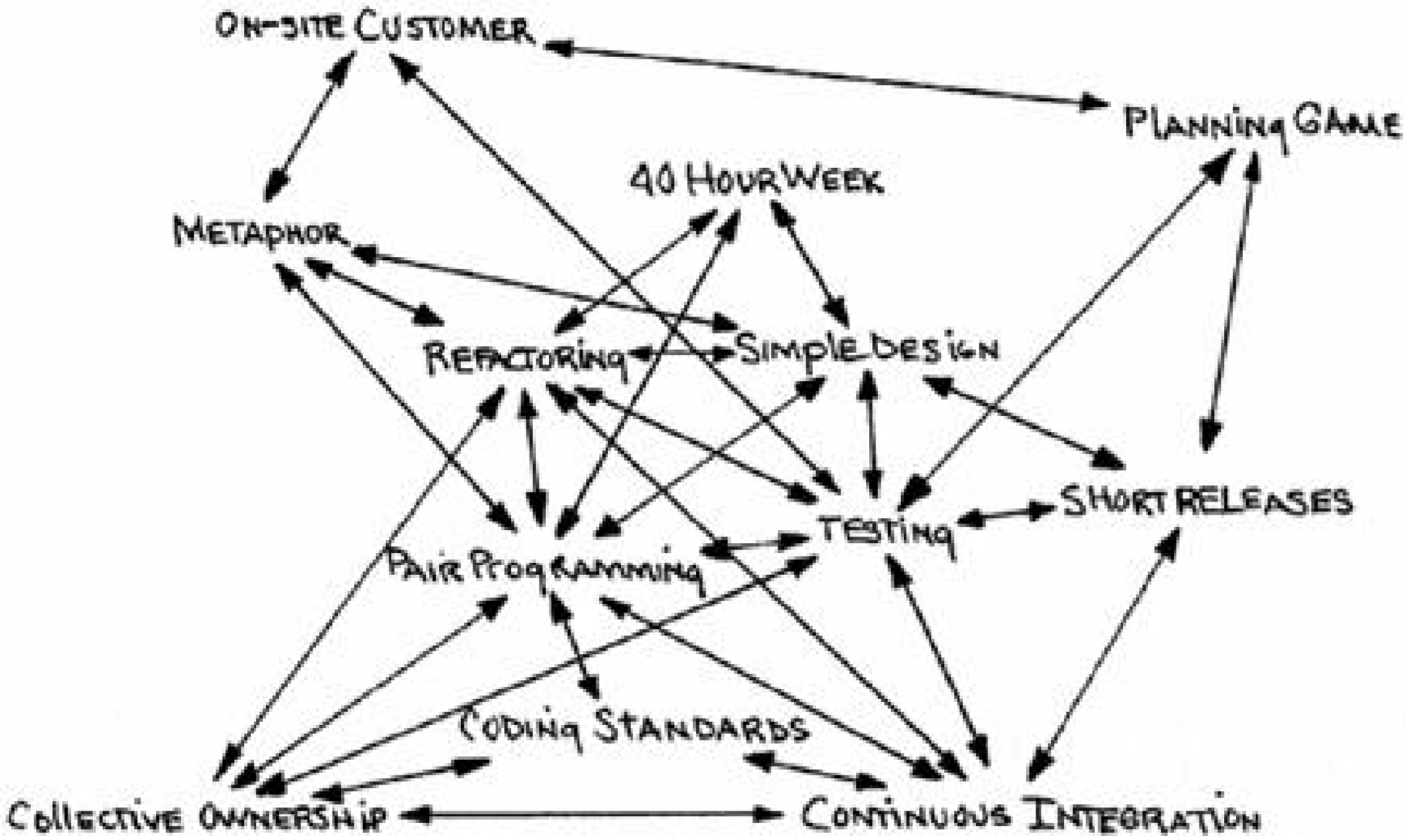
On-Site Customer

- In almost all **phases of XP** **communication with the customer** is necessary.
- Therefore, a **real customer** who is familiar with the problem domain (if possible an expert) and who is a potential user of the system is part of the **development team.**
- He or she will **sit with the team** and be available to provide additional information necessary to implement a **user story** (story refinement) answer questions and **resolve disputes** about the meaning of domain aspects set priorities at a small scale (the choice of stories for an iteration) write **functional tests and run** them at the end of the iteration (with help from the Tester)

Coding Standards

- XP projects use **rules and guidelines** for naming and formatting code units. This means, **every developer** chooses the names of **classes, methods, variables** etc. after these rules. The Coding Standards can come from the **outside or are defined by the development team**.
- The standards make the **system more consistent** so that it is easier to read, understand, and work with (extend, refactor).
- It is also a great help when somebody (perhaps a new team member) needs to **learn the system**.
- Therefore it is important that every team member follows the rules. If an **XP developer finds** some part of the system that doesn't follow the standards, he or she changes it accordingly (this, of course, needs Collective Code Ownership).
- Note that **Coding Standards** relate to the **System Metaphor.**

The Practices support each other



XP Roles

- eXtreme Programming defines the following roles
  - **Customer**
  - **Programmer**
  - **Coach**
  - **Tracker**
  - **Tester**

XP Roles

- **Customer**
- The customer **defines what to do** (User Stories) and in what order (Planning Games).
- XP has a special rule called **On-Site Customer.**
- **Programmer**
- XP is a **programmer-centric** methodology.
- It does not make use of **specialists like analysts, software architects or software designers**.
- Instead this work is performed by the programmers.
- There are **several skills that the programmer** must have
  - **Communication**
  - **Coding**
  - **Ability to work in team**

XP Roles

- **Coach**

The coach is **responsible** for the technical execution (and evolution) of the project.

- **Tracker**

The tracker will **ask each developer** every **two or three days** how much time they have spent on each of their tasks and **how much time is left.**

- **Tester**

The tester helps the **customer to choose** and write the **functional tests and run them** (especially the tests that cannot be automated).

XP Process

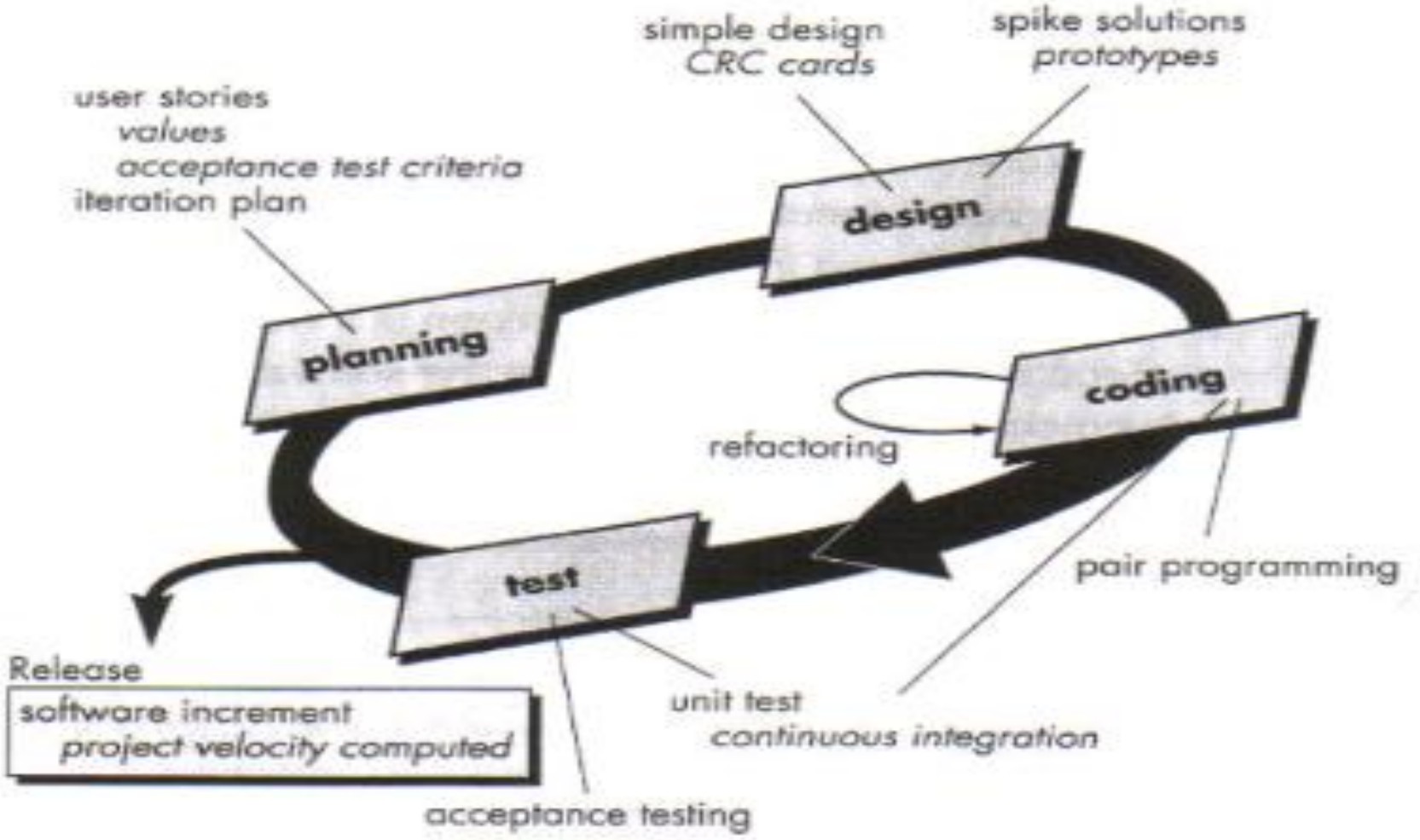


Fig. Extreme Programming Process

Planning

- The **first phase** of Extreme Programming life cycle is planning, where **customers or users** meet with the development team to create **'user stories'** or requirements.
- The **development team converts user stories into iterations** that cover a small part of the functionality or features required.
- A combination of iterations provides the customer with the **final fully functional product**.
- It includes

  Exploration,

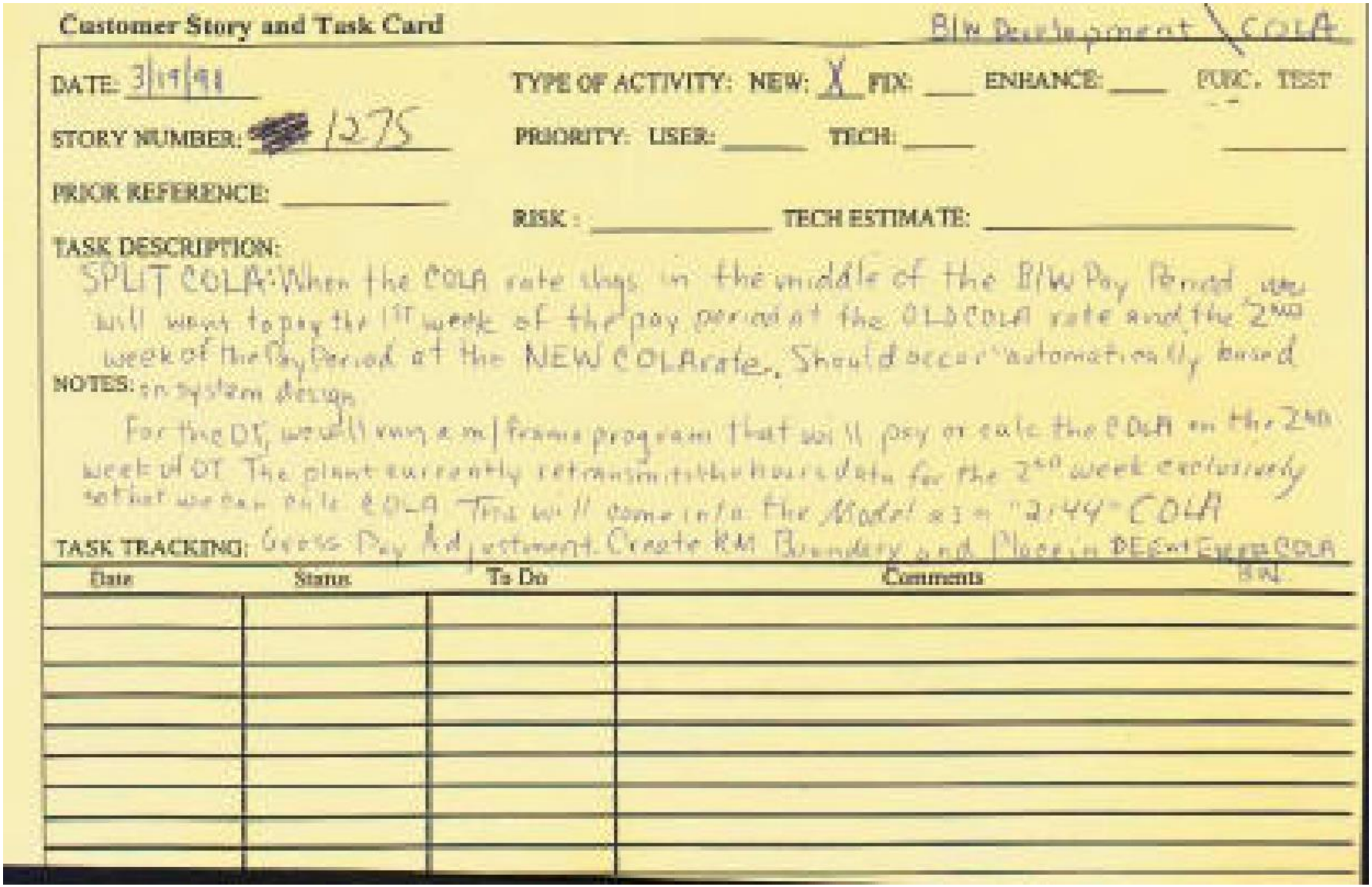
  Planning, and

  Steering
- In the **exploration phase** the customer **defines what** he/she **wants the system to do** and the developers estimate how **long it will take to implement** the desired behavior.

Planning

- In the **planning phase** both parties negotiate which of the desired features can be done within the given bounds (time, resources).
- **Steering** means **influencing** the process by little moves.
- It consists of four possible “moves” :
  - **Iteration**
  - **Recovery**
  - **New story**
  - **Re-estimation**

A User Story card from the C3 project



Customer Story and Task Card — B/W Development \ COLA

DATE: 3/19/98

TYPE OF ACTIVITY: NEW: X FIX: ___ ENHANCE: ___ PERF. TEST

STORY NUMBER: [redacted] 1275

PRIORITY: USER: ___ TECH: ___

PRIOR REFERENCE: ___

RISK: ___ TECH ESTIMATE: ___

TASK DESCRIPTION:

SPLIT COLA: When the COLA rate slips in the middle of the B/W Pay Period we will want to pay the 1st week of the pay period at the OLD COLA rate and the 2nd week of the Pay Period at the NEW COLA rate. Should occur automatically based on system design.

NOTES:

For the DL, we will run a mainframe program that will pay or rate the COLA on the 2nd week of DL. The plant currently returns to the hours data for the 2nd week exclusively so that we can rate COLA. This will come into the Model as a "2:44" COLA

TASK TRACKING: Guess Pay Adjustment. Create RM Boundary and Place in DEEM Exec COLA B/W

Date	Status	To Do	Comments

Designing

- An **iteration of XP** programming starts with **designing.**
- Using Software **Class Responsibilities and Collaboration (CRC)** Cards that allow for a departure from the traditional procedural **mindset** and make possible object oriented technology.
- Such cards allow all members of the **project team** to **contribute ideas**, and collate the best ideas into the design
- Creating **spike solutions or simple programs** that explore potential solutions for a specific problem, **ignoring all other concerns**, to mitigate risk

Coding

- Coding constitutes the most **important phase** in the Extreme Programming life cycle.
- XP programming gives **priority to the actual coding** over all other tasks such as **documentation** to ensure that the customer receives something substantial in value at the end of the day.
- Standards related to coding include:

✓ **Developing the code based** on the agreed metaphors and standards, and adopting a policy of **collective code ownership**.

✓ **Pair programming** or developing code by **two programmers** working together on a **single machine**, aimed at producing higher quality code at the same or less cost.

✓ **Strict adherence** to **40-hour workweeks** with no overtime.

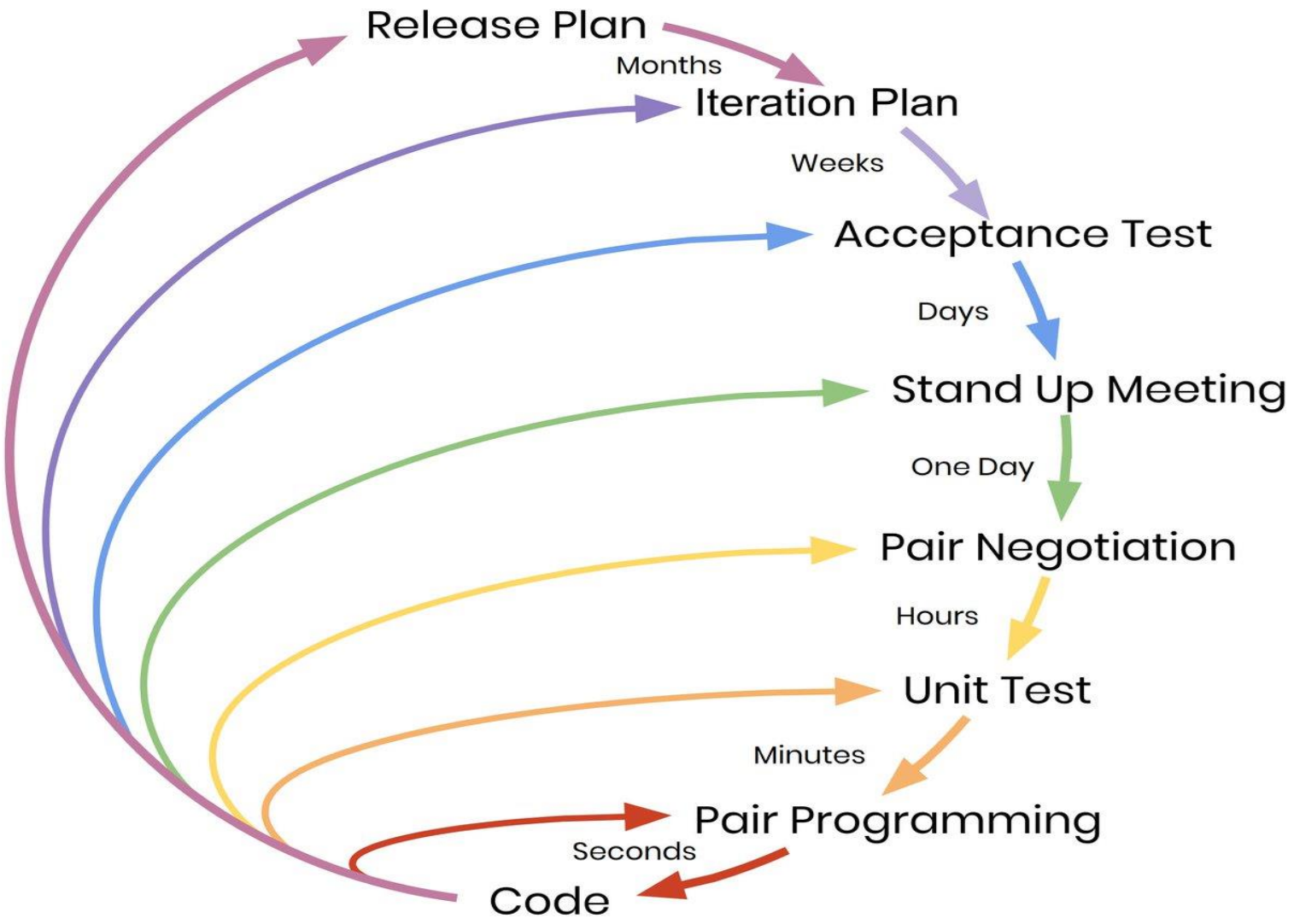
Testing

- Extreme program **integrates testing** with the development phase rather than at the **end of the development phase.**
- All **codes have unit tests** to **eliminate bugs**, and the code passes all such unit tests **before release**.
- Another **key test** is customer **acceptance tests**, based on the customer specifications.
- **Acceptance test run** at the completion of the coding, and the developers provide the customer with the results of the **acceptance tests** along with **demonstrations**.

Feedback Loops

- XP teams aim for a **very short iteration**—the **weekly cycle**—to increase feedback and shorten the feedback loop.
- To keep **feedback loops short**, they continuously reflect on how the project is going.
- If there's a **problem** then the team will talk about it. Osmotic communication from **sitting together** helps spread that feedback around the team.
- As they **communicate more** and get more continuous feedback, they start **seeing bottlenecks** in the way they're building software, and they **learn to work around obstacles together as a team.**

Feedback Loops



Feedback loops

Feedback Event	Feedback Duration
Pair Programming	Seconds
Unit Testing	Minutes
Clarifications among Team and with the Customer	Hours
Progress	In a Day
Acceptance Testing	Days
Iteration Planning	Weeks
Release Planning	Months

XP

Emphasises values like communication, simplicity, feedback and inclusiveness, with customer satisfaction as the pillar.

PROS

- **Simplicity**
  code is simple and allows for improvement
- **Visibility**
  Process and cycle are visible to developers
- **Agility**
  constant testing makes XP more agile
- **Encouragement**
  Uplifts and maintains talent in teams

CONS

- **Too focused**
  less importance paid to other aspects like design, because of extreme focus on code
- **Errors**
  lack of monitoring of errors may lead to similar bugs in the future

Lean

Origin:

- Management philosophy developed from **Toyota Production System** (TPS)
- The term "lean" has been applied to **manufacturing for many decades;** it was adapted for software development by Tom and Mary Poppendieck in the first decade of the **twenty-first century.**
- Use capital-**L term** "Lean" to refer to this adaption of lean ideas to **agile software development**.
- Lean is different. Unlike Scrum and XP, Lean doesn't include a set of practices. Lean is a **mindset**.
- The mindset of lean is sometimes called **lean thinking.**

Definition

A term Focused on **improving process** speed and quality through **reduction of process wastes**.

Lean thinking:

A business **methodology** which aims to provide a new way to think about how to organize human activities to **deliver more benefits** to society and value to individuals while **eliminating waste**.

Simple Definition:

"Becoming 'lean' is a process of **eliminating waste** with the goal of **creating value**."

Lean values

- **Eliminate waste**
- **Amplify learning**
- **Decide as late as possible**
- **Deliver as fast as possible**
- **Empower the team**
- **Build integrity in**
- **See the whole**

Eliminate waste

- Waste in lean is everything that **does not contribute to the value** for the customer, i.e. everything that does not help to **fulfill the needs of the customer** or does delight the customer.
- Seven types of waste were identified in manufacturing and mapped to software development.

  1.Partially done work **2. Extra processes**

  3. Extra features **4. Task switching**

  5. Waiting **6. Motion** **7. Defects**

- The wastes **slow down the development** flow and thus should be removed to speed up value creation.

Amplify learning

- Use **feedback** from your project to improve how you build software.
- In **every iteration**, there can be a new practice or learning to be gained, since we don't expect perfect work within the first delivery;
- Software development is a **knowledge-intensive process** where learning happens during the whole development lifecycle and needs to be amplified.
- Learning includes getting a **better understanding of the customer needs**, potential solutions for architecture, good testing strategies, and so forth

Decide as late as possible

- Make every important **decision for your project** when you have the most information about it—at the last responsible moment.
- Rather than making **early decisions and prone to error,** it's better to decide late after getting enough information for practices that involve uncertainty;
- A commitment should be **delayed as far as possible** for irreversible decisions.
- For example, a tough architectural decision might require some experimentation and therefore **should not be committed early.**
- Instead, the option for **change should be open** for as long as possible.

Deliver as fast as possible

- Lean has a strong focus on **short cycle times,** i.e. to **minimize the time from receiving a request** for a feature to the delivery of the feature.
- The reason for the strong focus on cycle time is that while a feature is **under development** it does **not create value** for the customer.
- Understand the **cost of delay,** and minimize it using pull systems and queues.
- Though speedy delivery customers can get **“what they need now”** rather than **“what they needed yesterday”** and moreover, it also helps to facilitate the learning process when deliver cycles are shorter;

Empower the team

- Establish a **focused and effective** work environment, and build a whole team of energized people.
- Involving workers in **key decision making** so that they know what need to be done by themselves and this is performed by using pull mechanism;
- The **three principles** that were used in the context of the Toyota Product Development System fostering the respect for people:

✓ **Entrepreneurial leadership**

✓ **Expert technical workforce**

✓ **Responsibility-based planning and control**

Build integrity in

- Build software that **intuitively makes sense to the users**, and which forms a coherent whole.
- Delivering a system that **exactly matches** and fulfills the user's wishes and needs; "Software with integrity has a coherent architecture, scores high on usability and fitness for purpose, and is **maintainable, adaptable, and extensible."**
- Quality of the software product should be **built in as early as possible,** and not late in development by fixing the defects that testing discovered.
- In result the integrity of the software in development should be **high at any point in time** during the development lifecycle.
- For example, if a defect is **discovered early** in the development process the ongoing work must be stopped and the **defect fixed.**

See the whole

- Implement **deep expertise** in every system area to enhance the **overall system performance.**
- Understand the work that **happens on your project**—and take the right kind of measurements to make sure you're actually seeing **everything clearly**, warts and all.
- When **improving the process** of software development the whole value-stream needs to be considered **end to end** (E2E).
- For example, there is no point in **sub-optimizing the requirements process** and by that increase the speed of the requirements flow into coding and testing if coding can only implement the requirements in a much slower pace.

Seven Wastes of Software Development

❖ **Partially done work**

- When you're **doing iteration**, you only deliver work that's **"done done"** because if it's not 100% complete and working, then you haven't delivered value to your users.
- Any activity that **doesn't deliver value** is **waste.**

❖ **Extra processes**

- **Processing beyond** the standard.
- All that extra **effort going through the process** of tracking and reporting time **doesn't create any value**.

Seven Wastes of Software Development

❖ **Extra features**

- **Creating more information** or tests than needed.
- When the **team builds a feature that nobody has asked** for instead of one that the users actually need, that's waste.
- Sometimes this happens because **someone on the team is very excited** about a new technology, and wants an opportunity to learn it.
- This might be valuable to the person who **improved their skillset,** and that might even be valuable in the future for the team.
- But it doesn't directly help build valuable software, s**o it's all waste.**

Seven Wastes of Software Development

❖ **Task switching**

- **Unnecessary movements between processes**.
- Some teams are **expected to multitask**, and often this multitasking gets out of hand.
- Team members feel like they already have a full-time job (like building software), plus many additional part-time tasks added on (like support, training, etc.)—and every bit of work is critical and top priority.
- The Scrum value of Focus helped us see that **switching between projects**, or even between **unrelated tasks** on the same project, adds unexpected delays and effort, because **context switching** requires a lot of cognitive overhead. Now we have another word for this: **waste.**

Seven Wastes of Software Development

❖ **Waiting**

- **People or items that wait** for a work cycle to be completed
- There are so many things that professional software developers have to **sit around and wait for**: someone needs to **finish reviewing a specification,** or approve access to some system that the project uses, or fix a computer problem, or obtain a software license... **these are all waste.**

❖ **Motion**

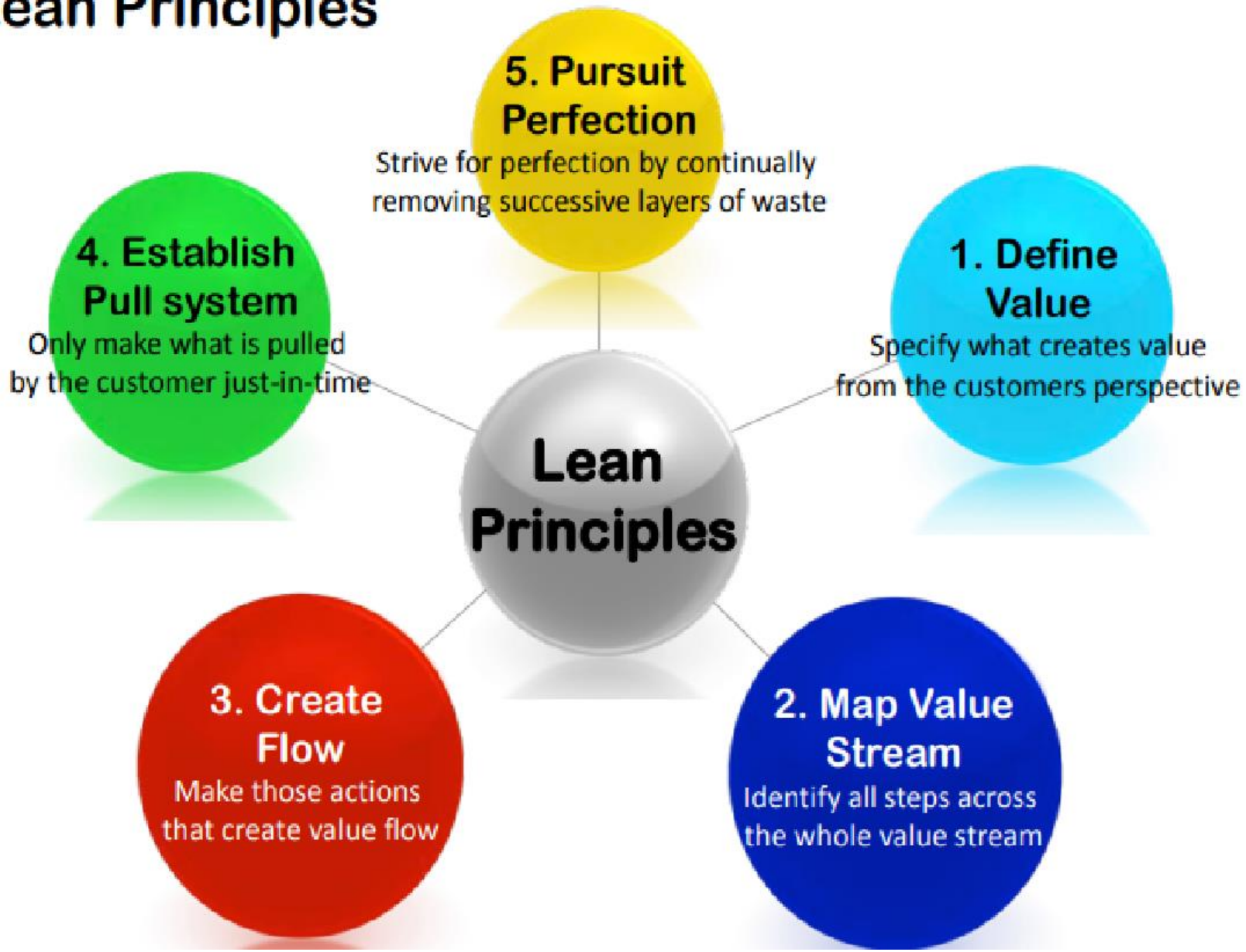
- **Unnecessary movement with a process**
- When the **team doesn't sit together**, people literally have to stand up and walk to their team members in order to have a discussion.
- This extra motion can actually add days or even weeks to a project if you add up all of the **time team members spend walking around**.

Seven Wastes of Software Development

❖ Defects

- **Test-driven development** prevents a lot of defects.
- Every developer who is **“test infected”** had that **“a ha”** moment where a unit test caught a bug that would have been very **painful to fix later**, and realized that it took a lot less time to write all of the tests than it would have to **track down that one bug**—especially if a user discovers the bug after the software is released.
- On the other hand, when the team needs to stay late **scrambling to fix bugs** that could have been prevented, that’s waste.

Lean Principles



Lean Thinking

Lean Principles		What should be done
1)	**Define Value**	Define value from a patient's perspective. Try to understand their health and non-health expectations. How patients' experience could be improved.
2)	**Map Value Stream**	Evaluate how all the steps of a process or procedure to provide services in the health facility. Then, eliminate any steps that do not contribute to performance, productivity or safety of the health facility.
3)	**Create Flow**	Eliminate waste between steps of a process and create smooth workflow for high efficiency
4)	**Establish Pull system**	Allow the patient to receive or request services if and when need.
5)	**Pursuit Perfection**	Continuously adapt to an ever-changing environment and patients' needs in order to deliver high quality of health services.

Lean Tools

- Just-in-time
- Kaizen
- Queuing Theory
- Leadership
- Cost of delay
- Value stream mapping

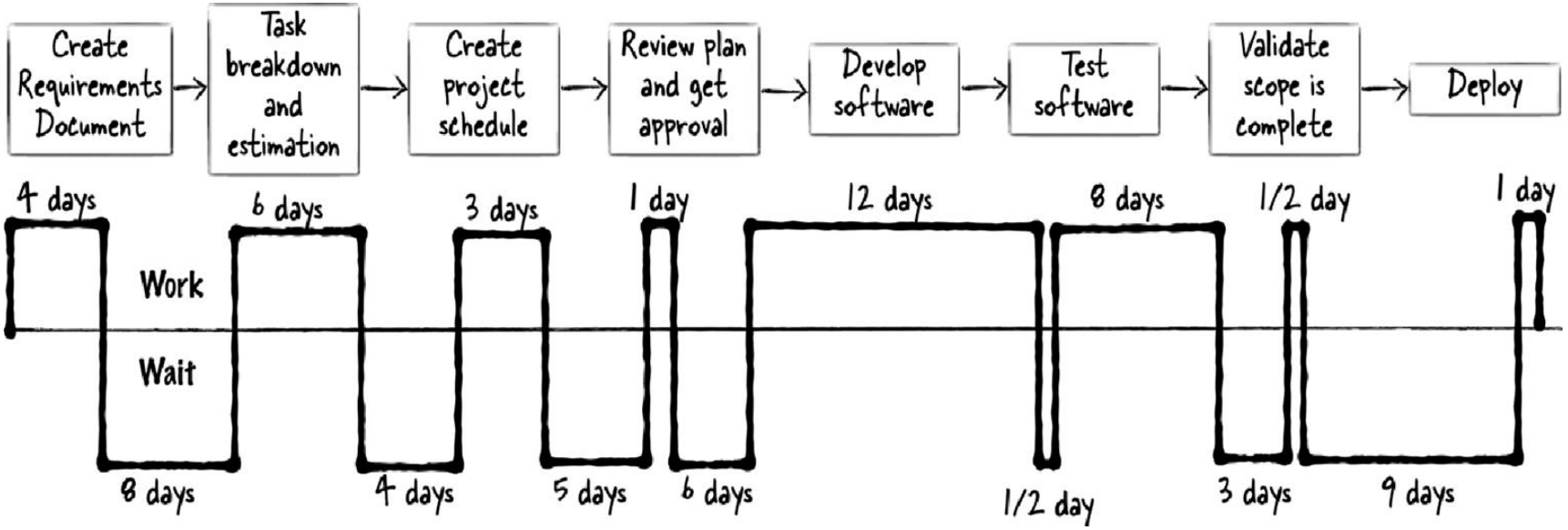
Value Stream Map

- In Lean Software Development, Mary and Tom Poppendieck recommend a simple **pencil-and-paper exercise to help you find waste**.
- It's called a **value stream map**, and you can build one for any process.
- Like many techniques used in conjunction with Lean, it originated in manufacturing, but it makes **sense for software teams as well.**

Steps in Value Stream Map

- Start with a **small unit of value** that the team has already built and delivered to the customers or users.
- Try to find the **smallest unit possible**— this is an example of a minimal marketable feature (MMF), or the smallest "chunk" of the product that the customers are **willing to prioritize**.
- Think back through all of the steps that the unit went through from **inception to delivery.**
- Draw a **box for every one of these steps**, using arrows to connect the boxes.
- Because you're drawing the actual path that a real feature took through your project, this will be a straight line—there are no decision points or forks in the path, because it represents the actual history of a real feature.
- An **MMF often takes the form of a user story, requirement, or feature request.**

Value Stream Map



- The value stream map clearly shows how much **wait time** was involved during the process.
- It took a total of **71 days** from the time the team started work on it to the time that it was deployed.
- Of those **71 days**, **35.5** were spent waiting rather than working.

Find the Root Cause of Problems

- Lean teams utilize a technique called **Five Whys** to figure out the root cause of a problem.
- **Why is the average lead time so long?**

Because it's taking over six months for most users' feature requests to make it into the software.

- **Why is it taking over six months for users' requests to make it into the software?**

Because those feature requests are almost always pushed back to make room on the schedule for last-minute changes.

- **Why are there so many last-minute changes?**

Because before the team can release software to the users, they need to do a review with senior managers, and those senior managers almost always ask for basic, foundational changes.

Find the Root Cause of Problems

- **Why do the senior managers almost always ask for basic, foundational changes?**

Because they all have very specific opinions about how the software should look, how it needs to function, and sometimes even the technical tools that should be used to build it, but the team doesn't hear those opinions until after they've built

all of the code and have demoed it to the senior managers.

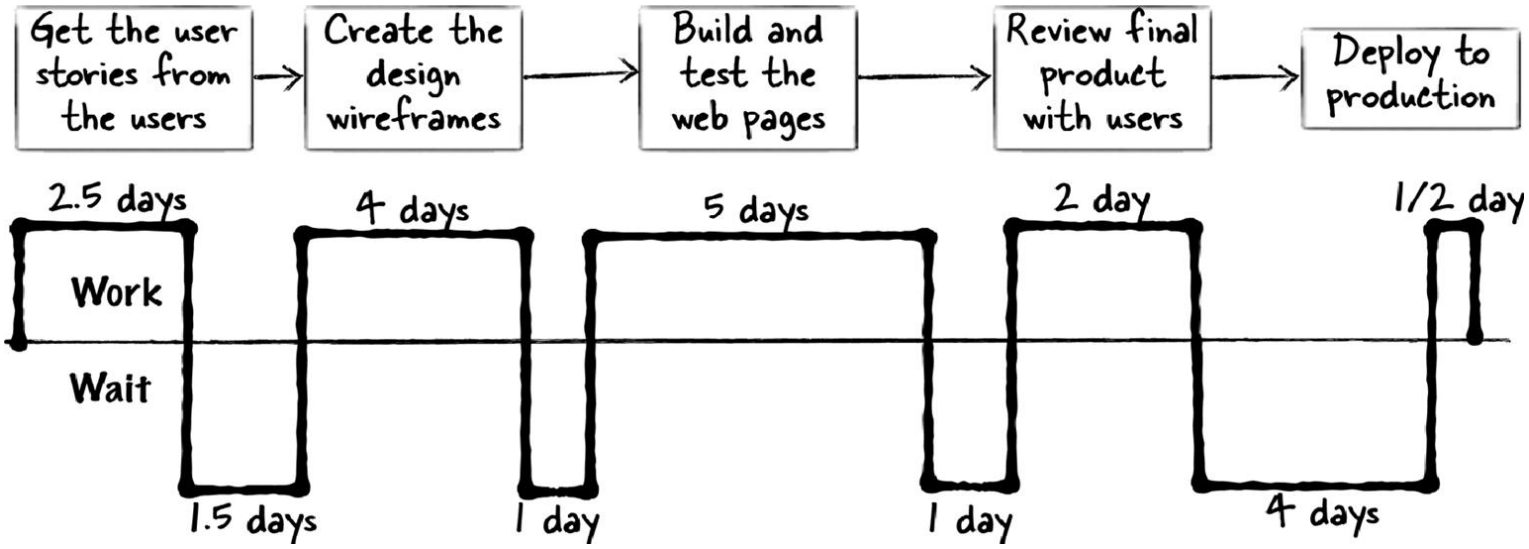
- **Why doesn't the team hear those opinions until after they've built all of the code and given a demo?**

Because the senior managers are too busy to talk to the team early in the project, so they'll only attend the final demo—and send the team back to the drawing board after the software is finished.

Area Chart to Visualize Work in Progress

- This is a simple diagram that shows how the **minimal marketable features** are flowing through your value stream.
- If you've created a **value stream map**, then you can build a WIP area chart, which is an area chart that shows how features, products, or other measures of value flow through **every part of the value stream**.
- This works best if you use MMFs, because they represent the **minimally sized "chunk"** of value that's created.
- The goal of the WIP area chart is to show a complete history of the work in progress —all of the **valuable features** that the team is currently working on.
- The chart will show us **how many MMFs** were in progress on any date, and how those MMFs broke down across the various value stream stages.
- The work in progress is a **measurement of features**

Area Chart

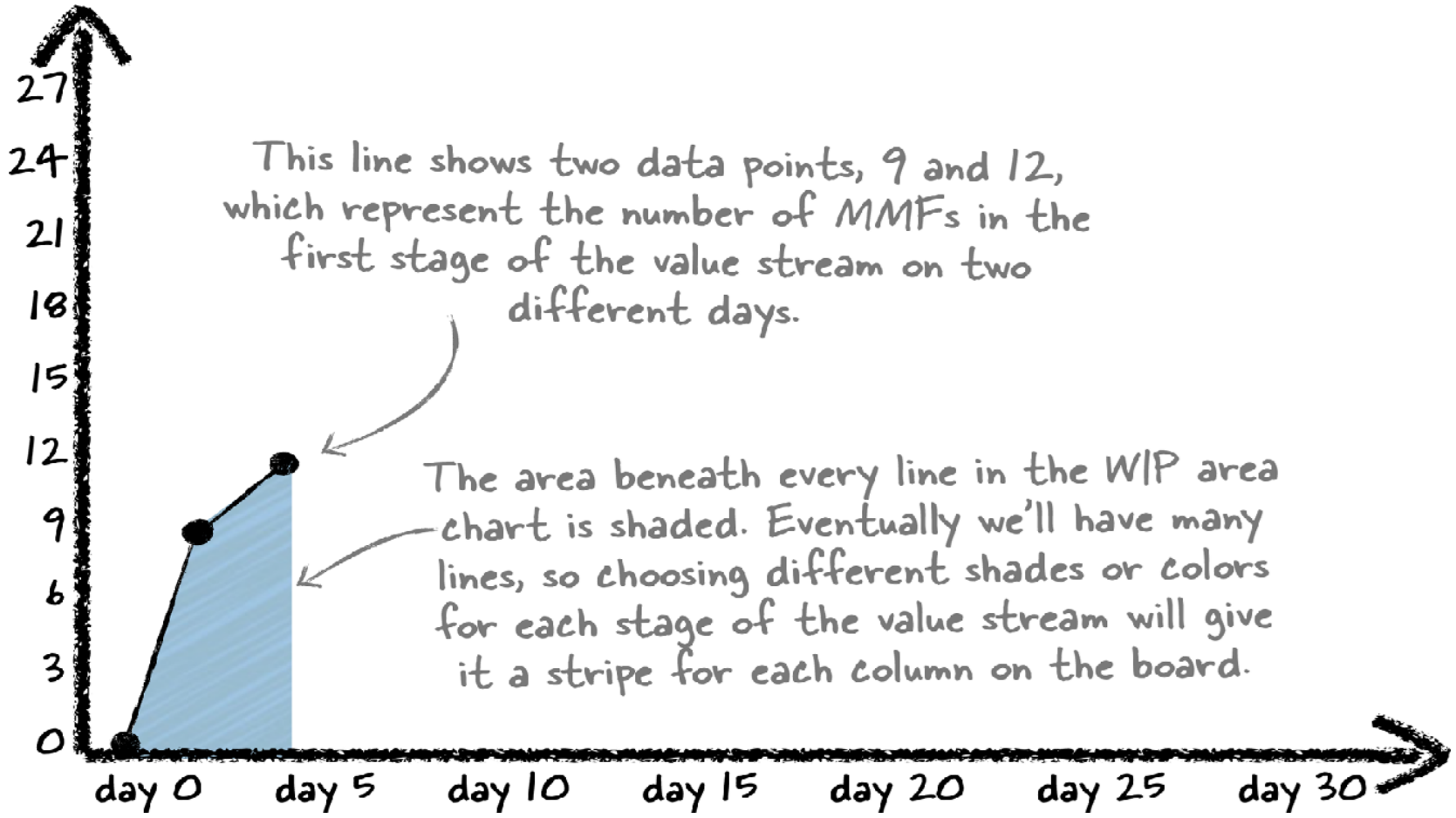


- We'll use this value stream map to build a **work-in-progress (WIP) area chart.**
- To build the WIP area chart, **start with an x-axis** that shows the **date**, and a **y-axis that shows the number of MMFs**.
- The chart has a line for each of the boxes in the value stream map.
- The lines divide the chart into areas for boxes in the value stream map.

Area Chart

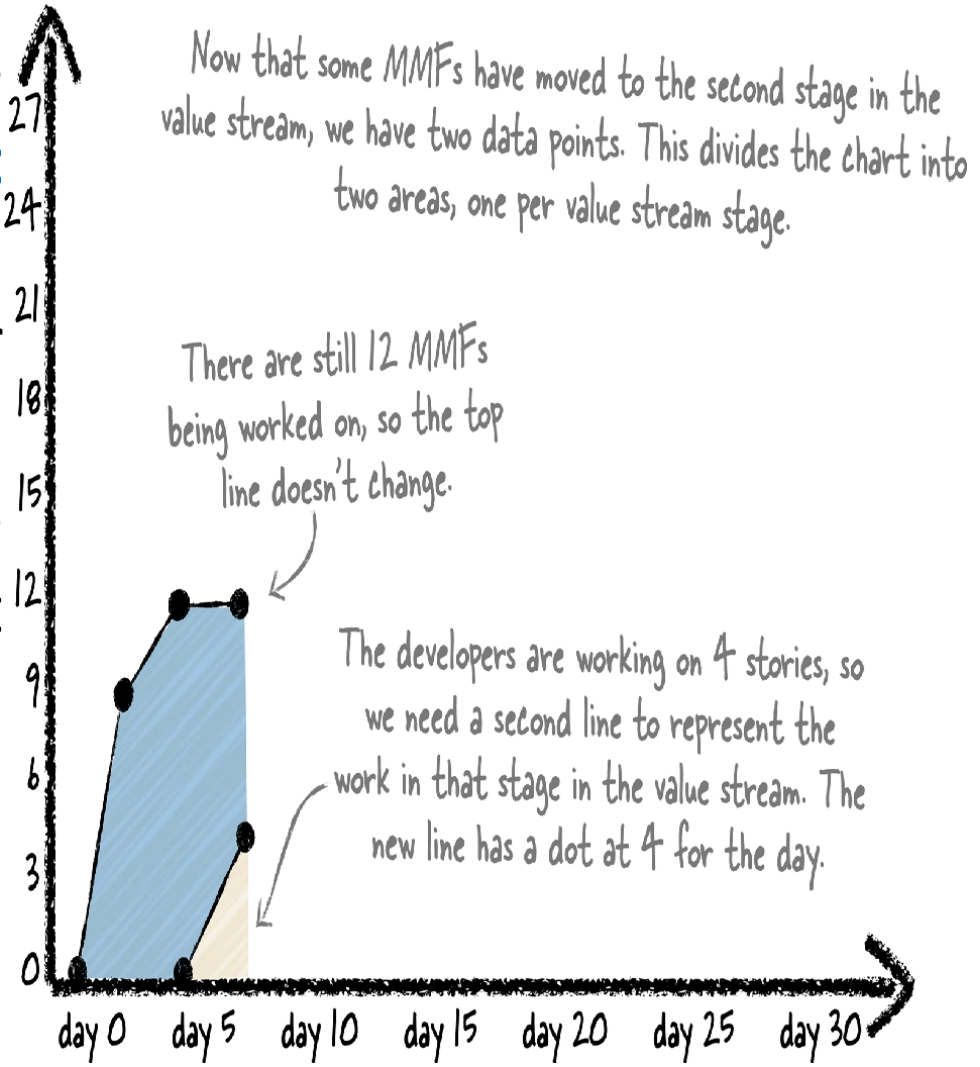
- There are no **MMFs in progress** before the project starts, so there's just a single dot at **X=0** at the lefthand side of the diagram (day 0).
- Let's say that when the **project starts**, the team starts working with the users on nine user stories, and they're using those **user stories as MMFs** for their project.
- Then a few days later, they add **three more stories**.
- You'll draw a dot at 9 for the first day, then another at **9 + 3 = 12** when those new MMFs were added, and you'll connect them.

Area Chart

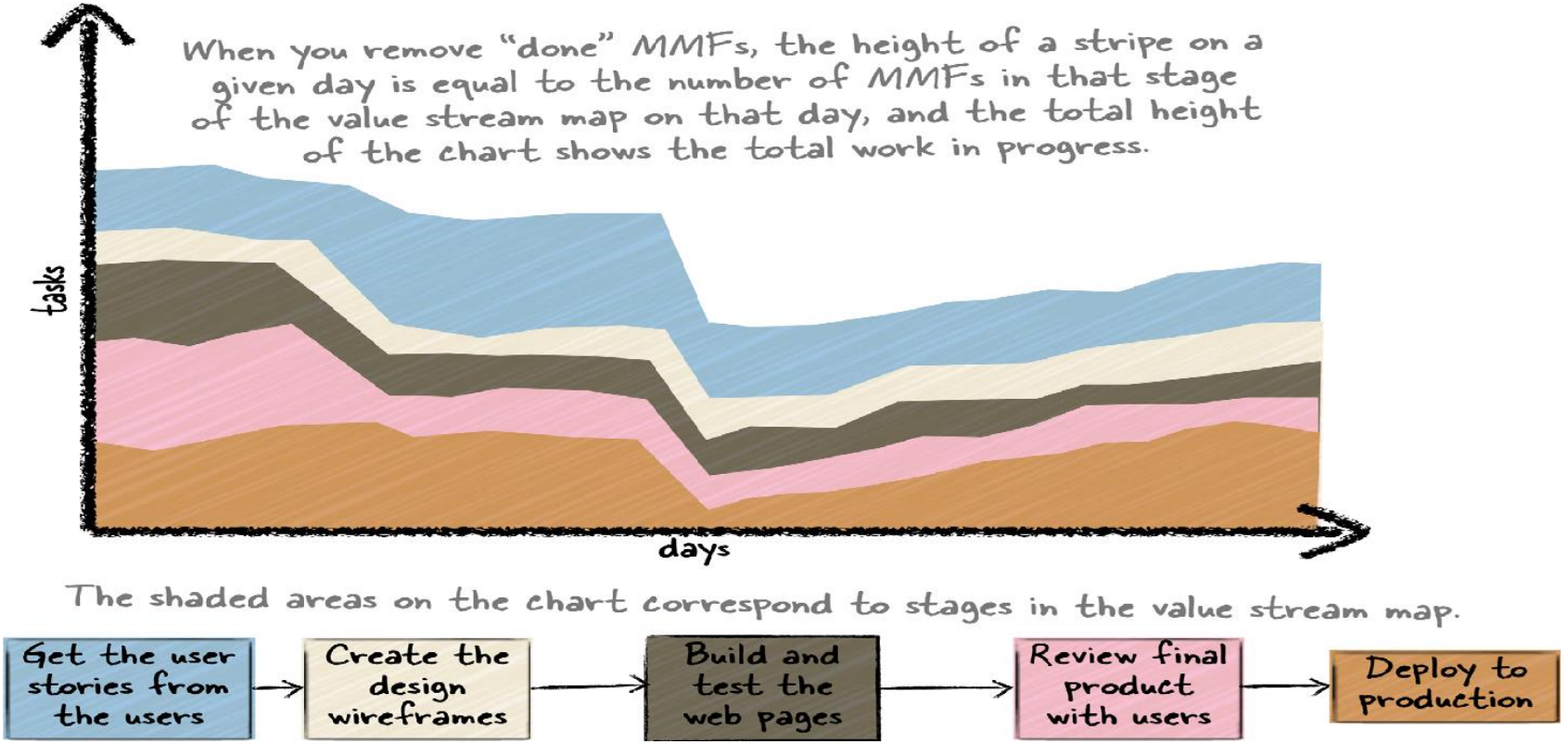

27
24
21
18
15
12
9
6
3
0
This line shows two data points, 9 and 12, which represent the number of MMFs in the first stage of the value stream on two different days.
The area beneath every line in the WIP area chart is shaded. Eventually we'll have many lines, so choosing different shades or colors for each stage of the value stream will give it a stripe for each column on the board.
day 0
day 5
day 10
day 15
day 20
day 25
day 30

Area Chart

- A few days later, the **programmers start** working on creating the design wireframes for four of the stories, so those **four stories** have progressed to the next stage in the value stream.



Area Chart



- It's useful to remove the **"done" tasks from the chart**, and use different shades for each stripe to make it easy to figure out which columns they correspond to.
- This is why most **WIP area charts do not include** completed tasks.

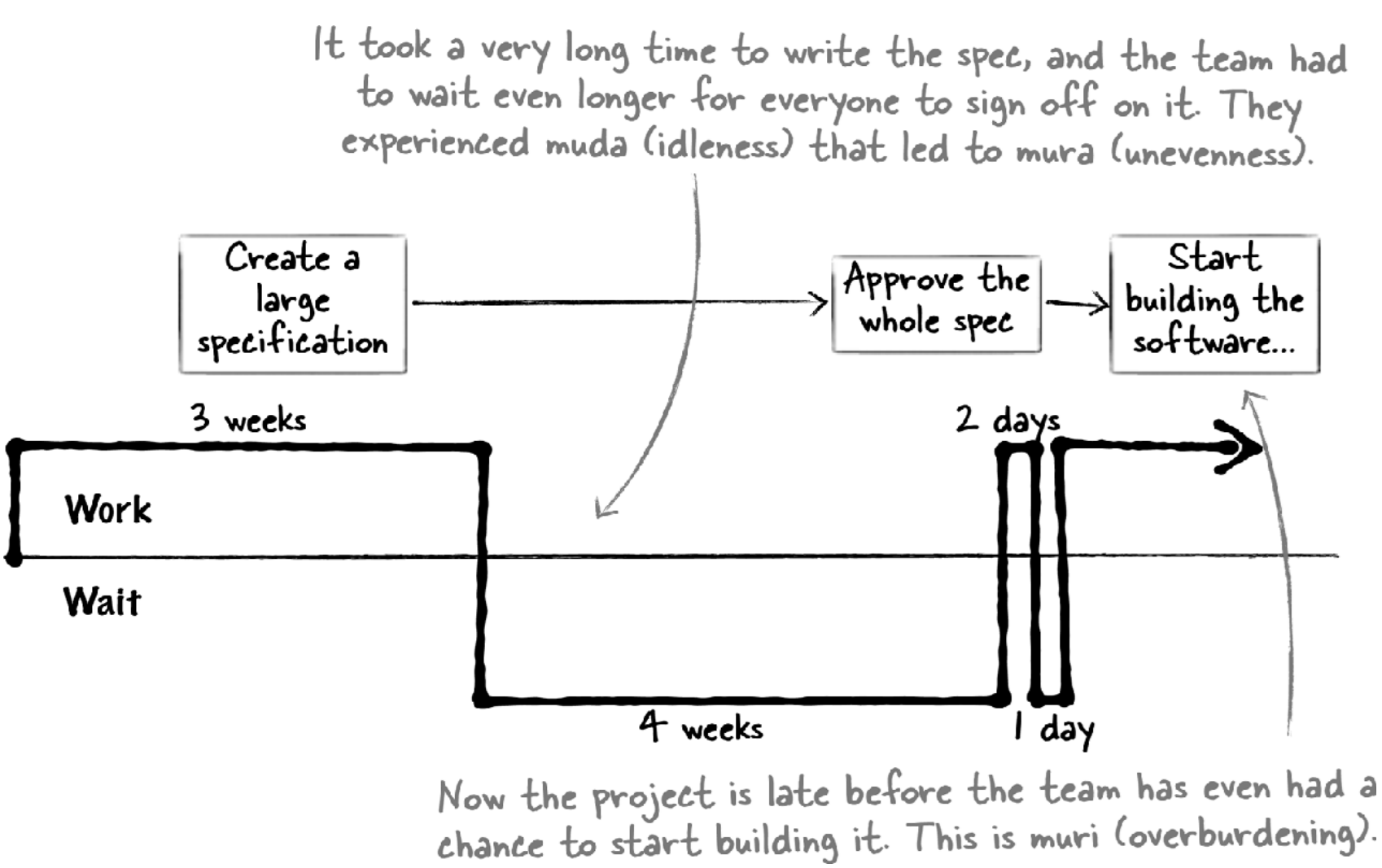
Pull Systems

- A pull system is a way of running a project or a process that uses queues or buffers to **eliminate constraints**.
- Pull systems are very useful for **building software**—maybe not surprisingly, for **exactly the same reason** that they're useful in manufacturing.
- Instead of having **users, managers, or product owners** push tasks, features, or requests down to a team, they'll add those requests to a queue that the team pulls from.
- When work backs up and causes unevenness partway through the project, they can **create a buffer** to help smooth it out.
- The team may have several different **queues and buffers** throughout their project.
- And as it turns out, this is a very **effective way of reducing waiting time, cutting out waste**, and helping users, managers, product owners, and software teams decide on what software is built.

3 wastes identified by Pull System

“Muri”, “Mura”, “Muda”

Muri Unreasonable burden	Any activity asking unreasonable stress or effort from personnel, material or equipment. In short: **OVERBURDEN** For people, Muri means: a too heavy mental- or physical burden. For machinery Muri means: expecting a machine to do more than it is capable of- or has been designed to do.
Mura Un-level workloads	Any variation leading to unbalanced situations. In short: **UNEVENNESS**, inconsistent, irregular. Mura exists when workflow is out of balance and workload is inconsistent and not incompliance with the standard.
Muda Any forms of Waste in the process	any activity in your process that does not add value. MUDA is not creating value for the customer. In short: **WASTE** Type I muda: Non-value-added tasks which seam to be essential. Business conditions need to be changed to eliminate this type of waste. Type II muda: Non-value-added tasks which can be eliminated immediately.



waste that happens when the team is waiting for a large specification to be created and approved.

Pull System

- A **pull system** is a better way to remove this unevenness and prevent the overburdening.
- The first step in creating a pull system is to **break the work** down into **small, pullable chunks**.
- So instead of building a **large spec**, the team can break it into **minimal marketable features**—say, individual stories, and maybe a small amount of documentation to go with each story.
- Those **stories** would then be **approved** individually.
- Typically, when a **spec review process** is held up for a long time, it's because people have problems with some of the features.

Pull System

- Approving individual **MMFs should allow** at least a few features to get approved quickly.
- As soon as the first MMF has been approved, the **team can start working** on it.
- Now the team **doesn't have to guess**. Instead, there's a real discussion of the work that needs to be done.

Kanban

- Kanban is a **method for process improvement** used by agile teams.
- Teams that use Kanban start by understanding the way that they currently build software, and make improvements to it over time.
- Like Scrum, **Kanban requires a mindset**.
- Kanban is a manufacturing term, adapted for software development by **David Anderson**.

Kanban

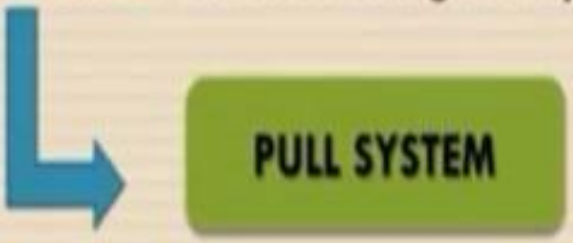
- Kanban → framework
- It is designed to help you
  - visualize your work,
  - work in progress,
  - work efficiency

  } Flow

- Kanban is a workflow management method.

History of Kanban

- KANBAN → Billboard or Signboard
- Initially, it used as a scheduling system for lean manufacturing, originating from the Toyota Production System (TPS).
- In the late 1940s, Toyota introduced "just in time" manufacturing to its production.



PULL SYSTEM

- A pull system allows you to begin new work only when there is either customer demand for it or goods are required by the next step within the production process.
- The pull system is also known as *built to order* production or inventory system.

The Original Kanban System

Kanban Values

Kanban Values

- **Transparency**

✓ By an **open exchange** of information and a clear and unambiguous vocabulary you create transparency in all areas.

- **Balance**

✓ You are efficient if you **balance the different requirements**, views and abilities of all participants among each other.

- **Collaboration**

✓ The Kanban method **improves the way people work together**. Collaboration is therefore one of its key points.

- **Customer Focus**

✓ The customers and the value (asset) they receive are the natural centre of interest of all persons involved in the company.

Kanban Values

- **Workflow**

✓ Work represents a continuous or occasional flow of values. An important starting point in using Kanban is to recognize and maintain such a flow of work.

- **Leadership**

✓ Leadership is required at all levels to generate value and achieve an improved state.

- **Understanding**

✓ Understanding means first and foremost self-awareness, both from the individual employee as well as from the entire organization to move forward.

✓ Kanban is a method of improvement and improvement requires change, which in turn requires understanding.

Kanban Values

- **Agreement**

✓ In an agreement, all parties agree to pursue goals together.

✓ Different opinions and approaches must be respected.

✓ These different points of view should converge

- **Respect**

✓ Respect for people in the form of appreciation, understanding and consideration is the foundation on which the other values are based

Kanban Principles and Practices

Kanban Principles

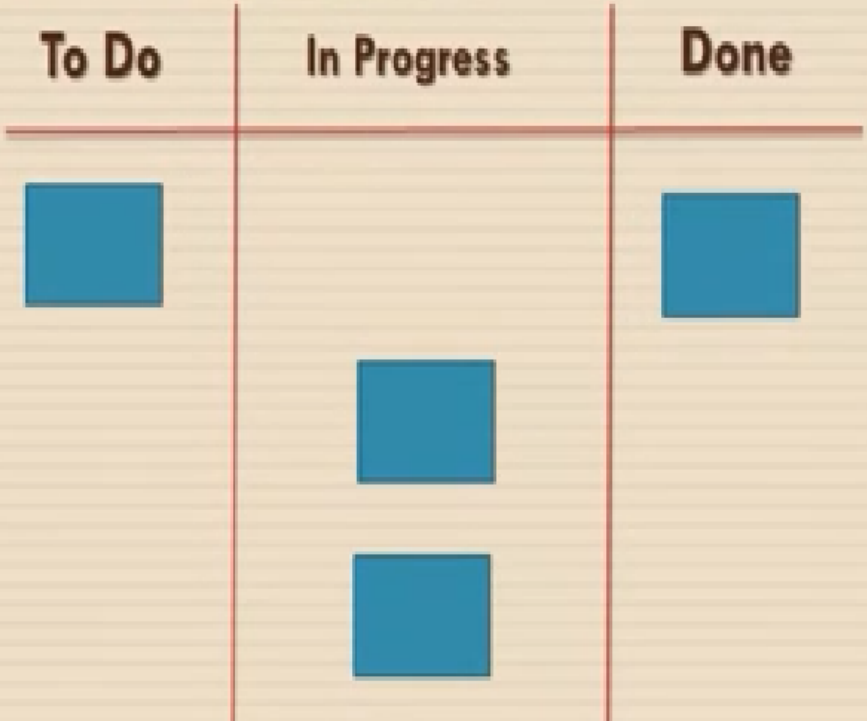
- Start with What You Do Now
- Agree to Pursue Incremental, Evolutionary change
- Respect the Current Process, Roles, Responsibilities & Titles
- Encourage Acts of Leadership at All Levels in Your Organization

Kanban Practices

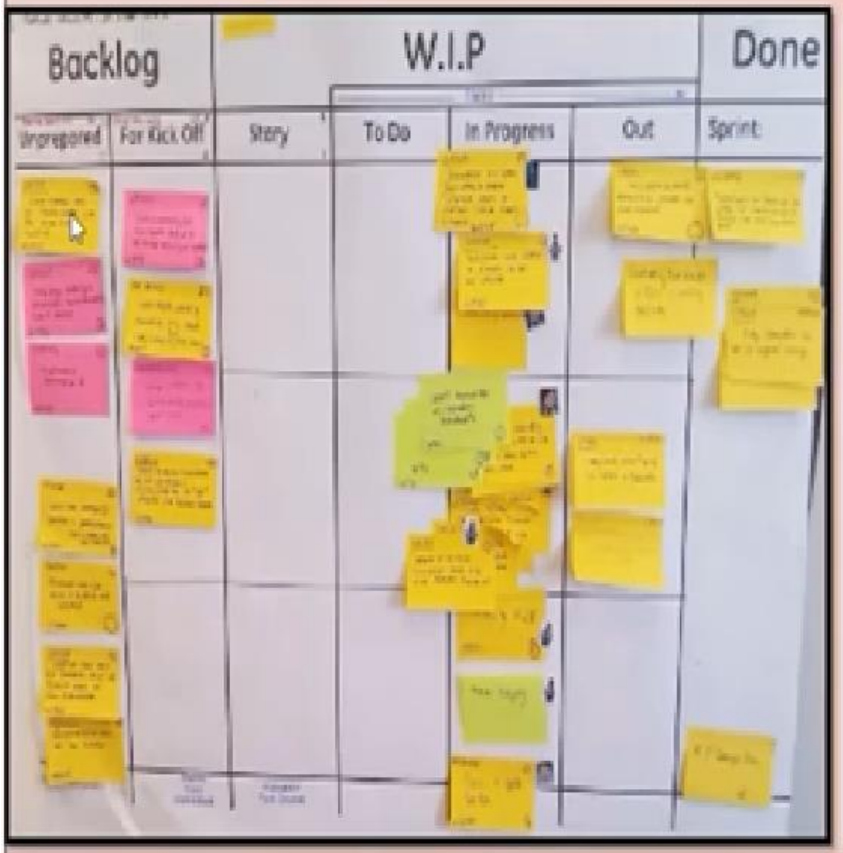
- Visualize the Workflow
- Limit Work In Progress
- Manage Flow
- Make Process Policies Explicit
- Implement Feedback Loops
- Improve Collaboratively & Evolve Experimentally

Kanban Boards

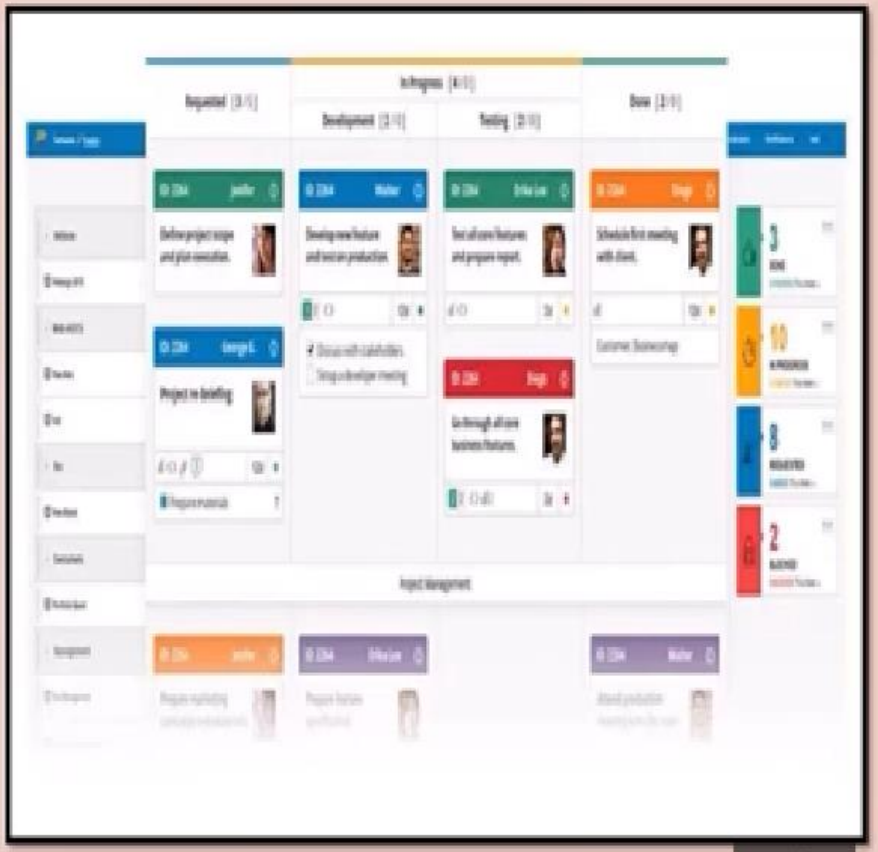
- A Kanban board is a tool for workflow visualization and one of the key components of the Kanban method.
- A basic kanban board has a three-step workflow:



KANBAN BOARDS

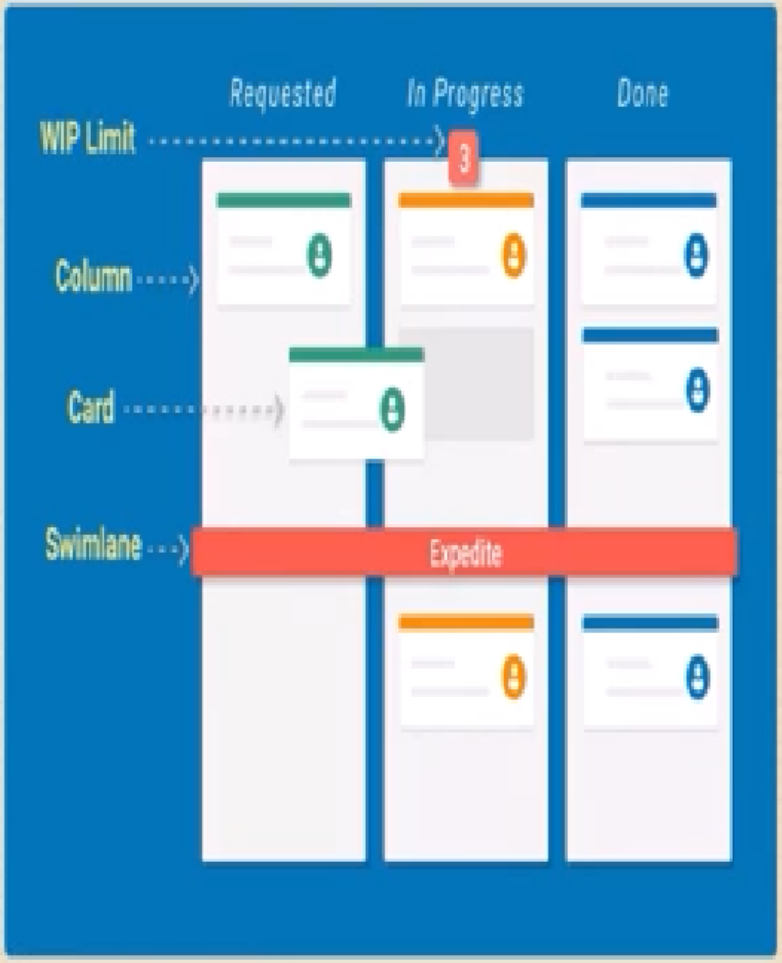


Virtual board



Digital board

KANBAN BOARD



- Components of the Kanban board

(i) Kanban Cards

(ii) Kanban Columns

(iii) Work-in-Progress Limits

(iv) Kanban Swimlanes

Kanban Cards

- In Japanese, Kanban → visual signal

- A Kanban card is a visual representation of a work item.
- A Kanban card contains valuable information about the task and its status.
- It ensures increased focus, full traceability, and fast identification of blockers and dependencies
- Record crucial documentation for a task
- Review details at a glance
- Minimize waste time
- Identify opportunities for collaboration

Visualize the Workflow

- The first step in improving a process is understanding how the team currently works, and that's what the Kanban practice visualize is about.
- In Kanban, visualizing means writing down exactly what the team does, warts and all, without embellishing it.

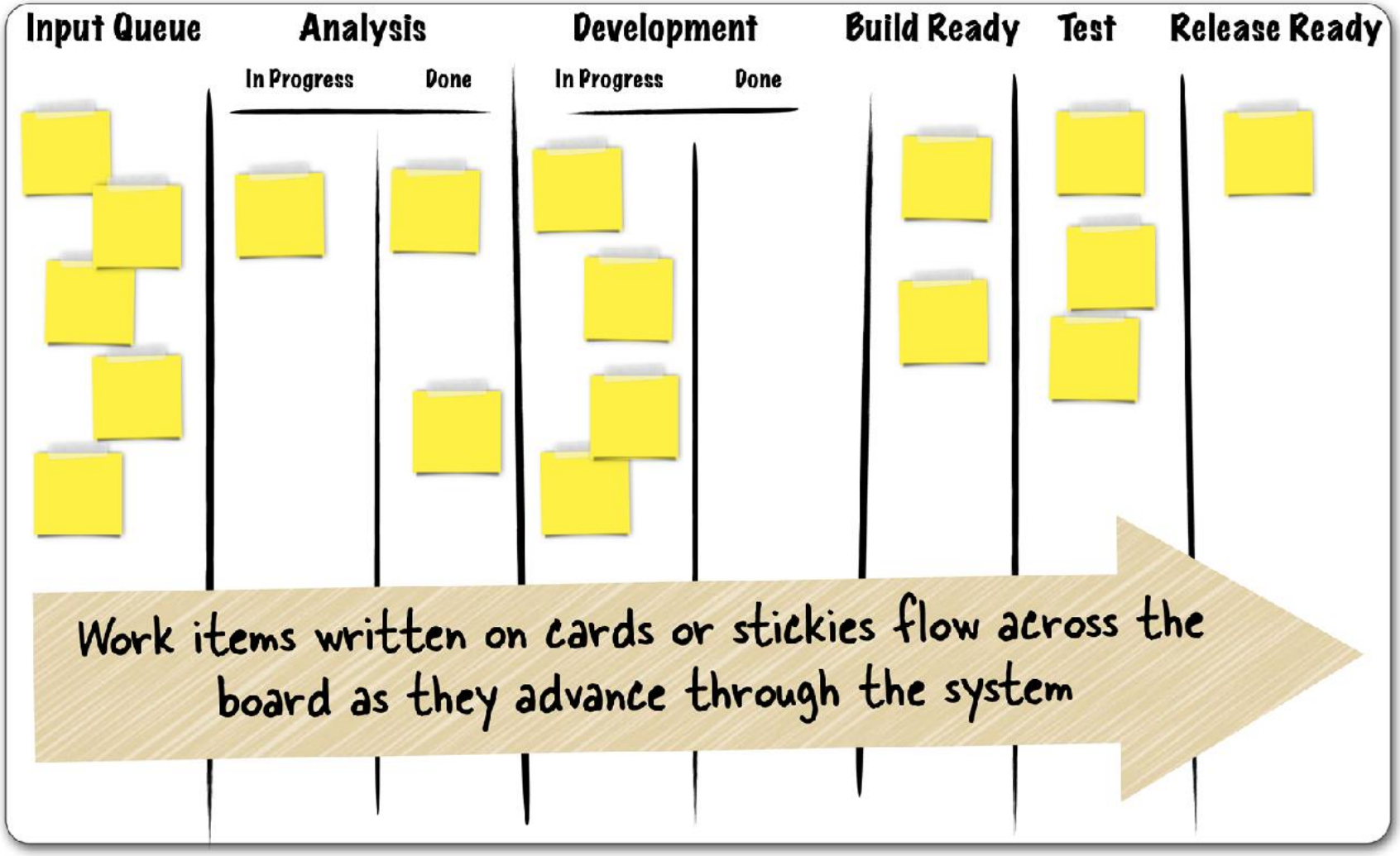
Use a kanban board to visualize the workflow

- A kanban board is a tool that teams use to visualize their workflow. (The K in the methodology name Kanban is typically uppercase; the k in kanban board is usually lowercase.)
- A kanban board looks a lot like a Scrum task board: it typically consists of columns drawn on a whiteboard, with sticky notes stuck in each column. (It's more common to find sticky notes stuck to kanban boards than it is to find index cards.)
- There are three very important differences between a task board and a kanban board.
- You already learned about the first difference: that kanban boards only have stories, and do not show tasks.
- Another difference is that columns in kanban boards usually vary from team to team. Finally, kanban boards can set limits on the amount of work in a column.

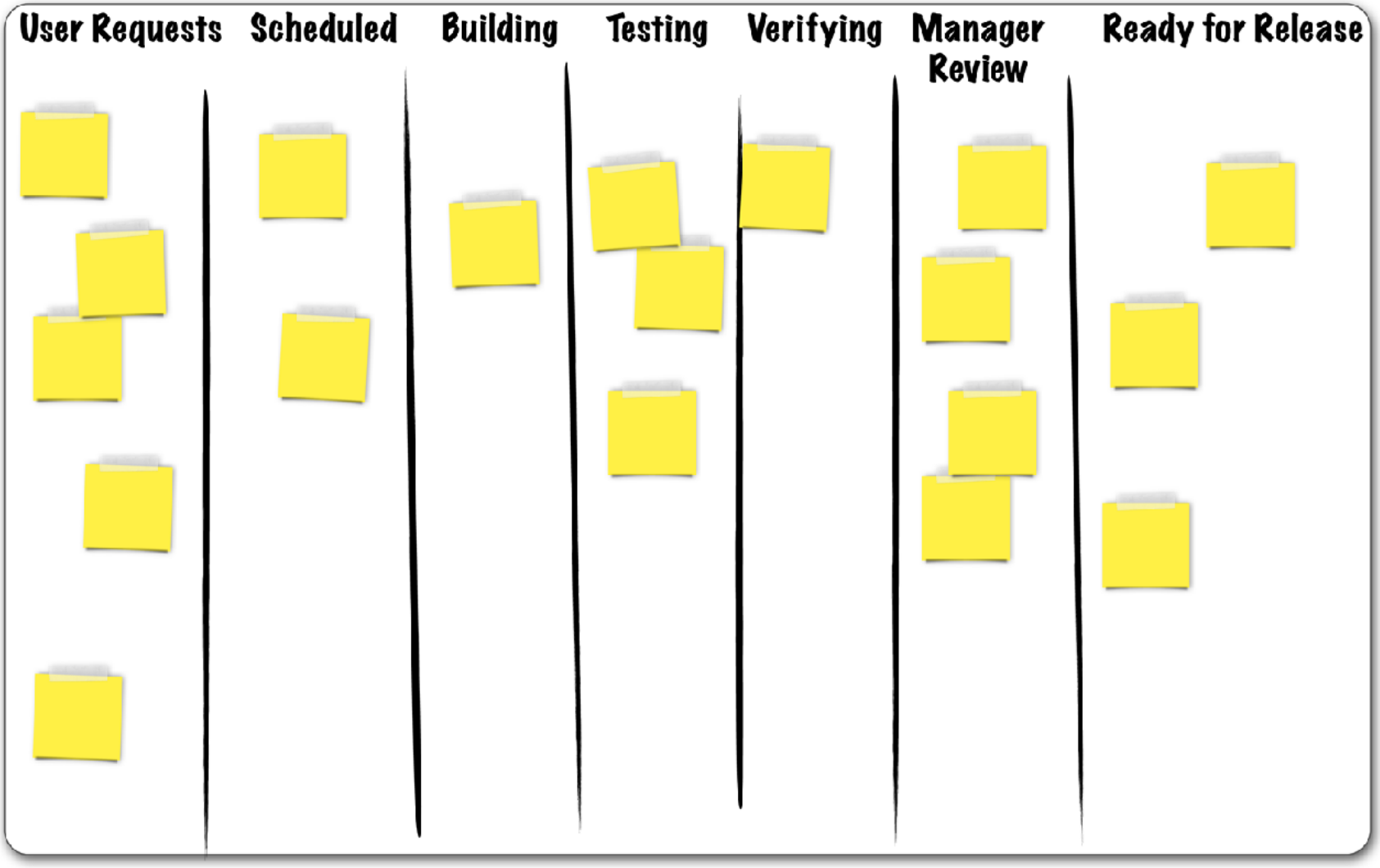
Visualize the Workflow

- When a team wants to adopt Kanban, the first thing that they do is visualize the workflow by creating a kanban board.
- For example, one of the first kanban boards in David Anderson's book, Kanban, has these columns: Input Queue, Analysis (In Prog),
- Analysis (Done), Dev Ready, Development (In Prog), Development (Done), Build Ready, Test, and Release Ready. This board would be used by a team that follows a process where each feature goes through analysis, development, build, and test.

Kanban Board



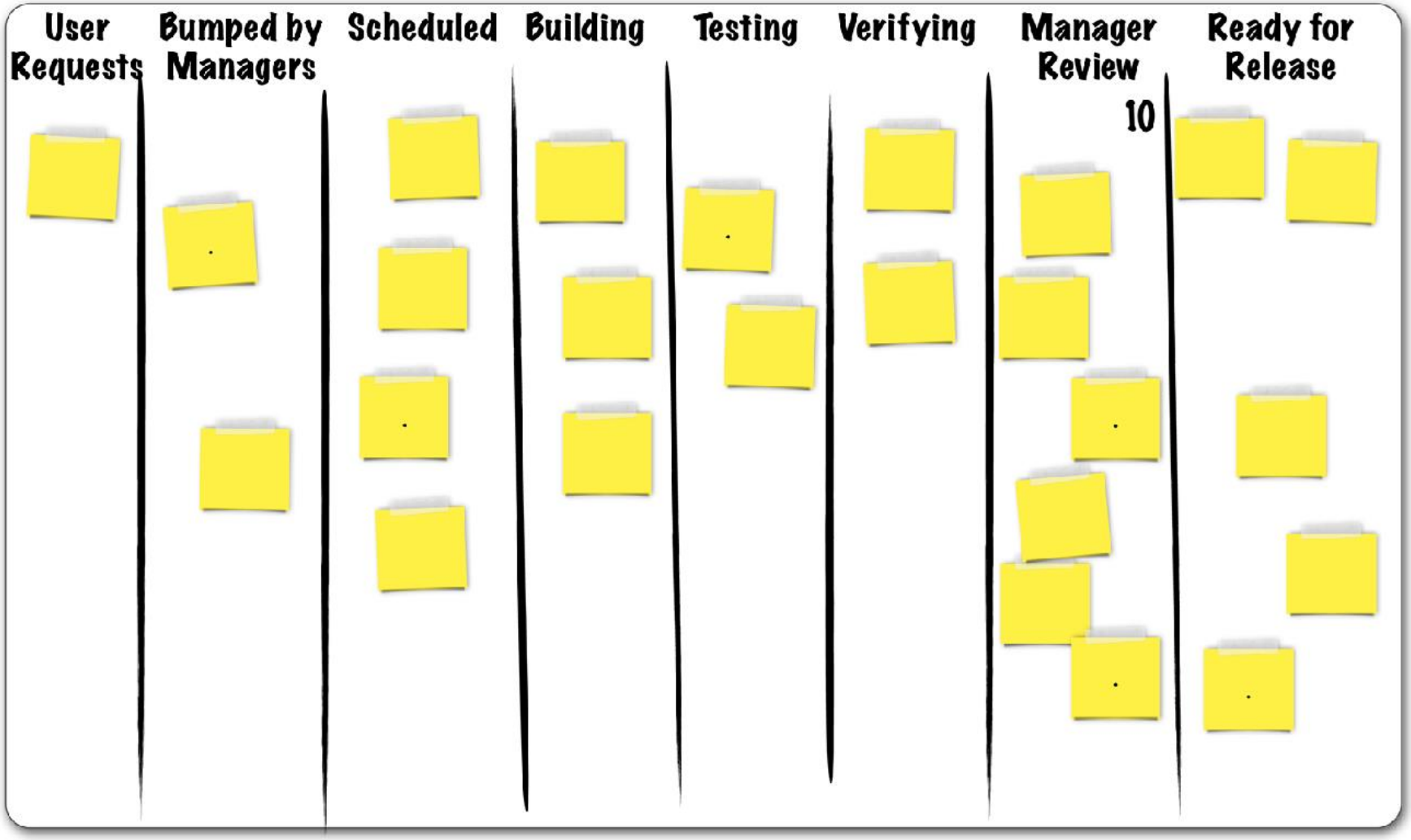
This kanban board gives a more accurate and realistic picture of how theproject is run.



Limit Work in Progress

- A team can only do so much work at a time. We learned this with Scrum.
- When a team agrees to do more work than they can actually accomplish by the time they'd agreed to deliver it, bad things happen.
- They either leave some of the work out of the delivery, do a poor job of building the product, or work at an unsustainable pace that will cost dearly in future releases.
- Limiting work in progress (WIP) means setting a limit on the number of work items that can be in a particular stage in the project's workflow.

The number 10 in the Manager Review column is its WIP limit

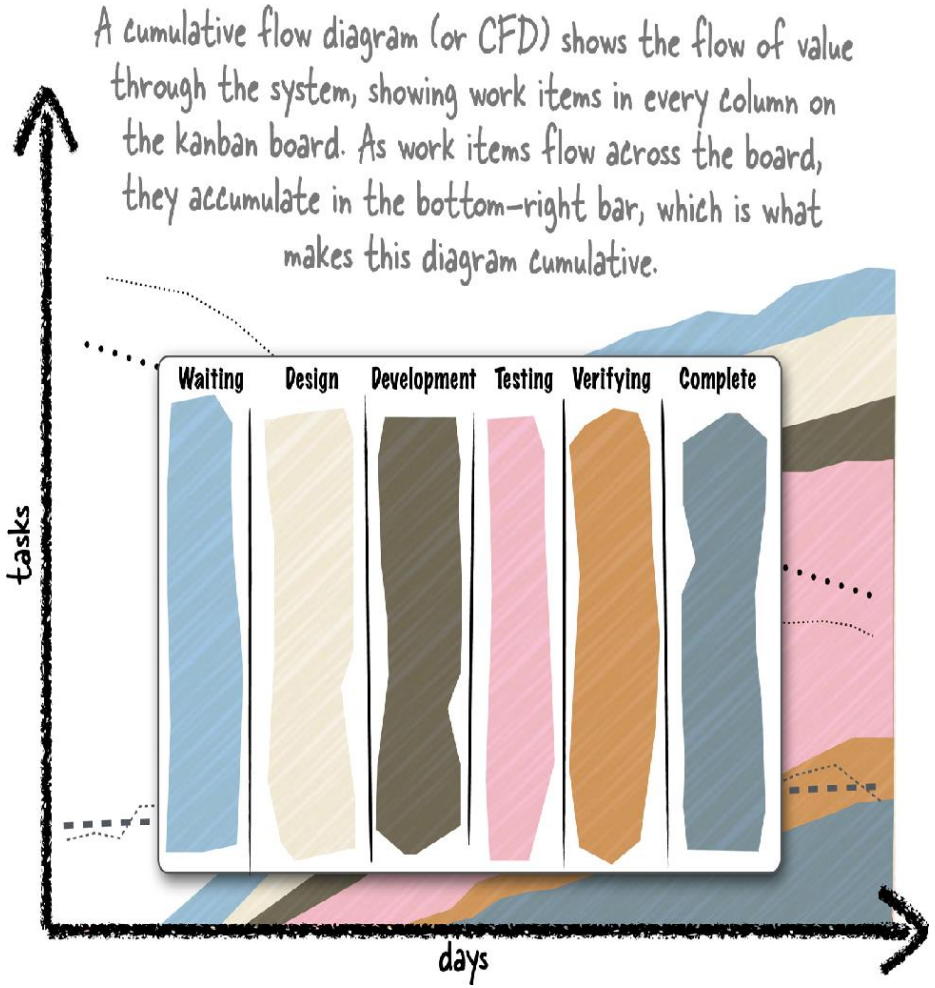


Measure and Manage Flow

- As teams continue to deliver work, they identify workflow problems and adjust their WIP limits so that the feedback loops provide enough information without causing thrashing.
- The flow of the system is the rate at which work items move through it.
- When the team finds an optimal pace for delivery combined with a comfortable amount of feedback, they've maximized the flow.
- A Kanban team uses the manage flow practice by measuring the flow and taking active steps to improve it for the team.

Measure and Manage Flow

- The kanban board is an important tool for managing flow specifically because it visualizes the source of the problem, and lets you limit work in progress where it will be most effective.
- An effective tool for measuring flow is a cumulative flow diagram, orCFD.



Make Process Policies Explicit

- Kanban teams don't need to write long documents or create huge Wikis to establish explicit policies.
- Policies can be as simple as WIP limits at the tops of the columns.
- Teams can also write down their policies by adding "definitions of done" or "exit criteria" bullets to the bottom of each column on a kanban board, so that the team members know exactly when to advance the work items through the workflow.
- This is especially effective when these policies were established collaboratively and evolved experimentally by the whole team, because it means that everyone understands why they're there.
- Complex processes and unwritten rules tend to emerge over time, and they seem to be especially common on teams that have a fractured perspective.

DESUVIT

KANBAN VS SCRUM

CATEGORY	KANBAN	SCRUM
Roles	No fixed roles. There might be a Project Manager, but team collaboration is the focus.	Three fixed roles. Scrum Master, Product Owner, and development team.
Due Dates	Continuous delivery	At the end of each sprint
Changes	Follows an adaptive methodology	Changes mid-work are discouraged.
Key metrics	Lead time, cycle time, and WIP	Velocity
Best Applications	Projects with different priorities	Teams with stable priorities not subject to change
WIP	Limits WIP; focus on the flow of work	No WIP limits; focus on completing work in the timebox.