Grooming and Planning

Introduction

- As the development process progresses, the requirements/user stories are groomed into functional input for the development teams.
- The priority of the user stories and the estimates of their size are important factors when the development team is considering their commitment to the work.
- Once the grooming is complete, the sprint planning begins, incorporating the velocity of the team, the definition of “done,” the amount of technical debt, and more.
- Finally, we explore the theory of triple constraints (scope versus time versus resources) and how the management of these can affect the development process.

Scrum and XP – Product Backlogs

- The full list of desired features, or user stories, are captured in what Scrum and XP call a “backlog.”

- The highest priority stories reside at the top of the backlog and are in the lowest level of detail.
- Stories that are deeper in the backlog, meaning they will not be worked on for some time, will likely be in epic form, which is perfectly acceptable.

Scrum and XP – Product Backlogs

- Backlogs vary in their depth, breadth, and quality.

- The acronym DEEP, which stands for **D**etailed appropriately, **E**stimated, **E**mergent, and **P**rioritized provides sound guidance on how to improve backlog quality.

Scrum and XP – Product Backlogs

- **Detailed appropriately** means that the highest priority stories contain sufficient detail for the development teams to deliver them: Questions are answered, and the necessary clarifications are included in the story description. Acceptance criteria are a component of the essential detail for the story.

- **Estimated** means that the team understands the stories and believes there is adequate information to estimate the level of effort or amount of time required to deliver the story.

Scrum and XP – Product Backlogs

- **<u>E</u>mergent** refers to the backlog's constant evolution: As new information is learned about the marketplace, a competitor, or a technological advancement, the backlog is modified, reprioritized, or clarified.
- **<u>P</u>rioritized** means exactly that—the user stories are in priority order, with the highest priority items that will deliver the most business value at the top of the backlog for immediate development.

Working Agreement

The Team – Working Agreement

- One way that trust and teamwork are established and enforced is through the creation of a working agreement.

- This is a document or set of expectations that define how the Scrum team is going to work together.

- The working agreement is the first point of collaboration for a new Scrum team as they define their relationships.

- It is more than just rules of engagement for team behavior; the working agreement ultimately reflects the values and commitment of the team.

The Team – Working Agreement Topics

- Time and location of Daily Scrum
- Testing strategy (unit, functional, integration, performance, stress, etc...)
- Build and infrastructure plans (shared responsibilities)
- Team norms (be on time, respect estimates, help when needed, etc...)
- How to address bugs/fires during Sprint
- Product Owner availability (phone, office hours, attendance in Daily Scrum)
- Capacity plan for initial Sprint(s)

Prioritization of Stories

Prioritization of Stories

- Product backlog requirements/stories must be prioritized.

- There are a number of tools and methods to assist with this activity:
  - Prioritization Based on Value
  - Value Stream Mapping
  - MoSCoW
  - Kano Model

Prioritization Based on Value

- We have previously discussed the concept of prioritization based on business value.

- Business value can take many forms:
  1. Increased revenue
  2. Expansion of addressable market (i.e., with this new feature, more people will be interested in buying it)
  3. Decreased cost
  4. Increased customer satisfaction
  5. Increased processing speed
  6. Increased stability of the application
  7. Improved usability

Prioritization Based on Value

- Value must also be supported by a positive return on investment (ROI), so the cost to deliver a desired feature must be considered.

- If there is no way to recoup your investment because the cost of delivering the feature is so high, then the value deteriorates and the feature should be deprioritized.

Prioritization Based on Value

- Also there is a consideration of the risks associated with the feature development:
  - If this feature might jeopardize the stability of the application, then its value is diminished.

- You can see that value is not a simple equation and that many factors—revenue, cost, risk, and penalties—must be considered.

Value Stream Mapping

- Value stream mapping is designed to remove waste from the product development cycle.

- The mapping begins and ends with customer activity, and the entire process is viewed from the customer’s perspective; this allows organizations to find breakpoints or bottlenecks in their process, where the customer experience is less than optimal.

Value Stream Mapping

- Sometimes this happens because the organizational processes get in the way.

- For example, have you ever talked to a customer service representative about a billing question and then wanted to change your mailing address, only to be told that the person you are talking to cannot handle that additional transaction?

Value Stream Mapping

- This happens when a company has developed organizational silos without regard to the customer impacts of those silos.

- The concept of value stream mapping is the flow all of the transactions from the eyes of the consumers to ensure that their experience is seamless and optimized.

MoSCoW

- Under this method, the “must haves” would clearly take priority over the “should haves,” and so on.

- The elements of MoSCoW are:
  - **Must have:** All features classified in this group must be implemented, and if they are not delivered, the system would simply not work.
  - **Should have:** Features of this priority are important but can be omitted if time or resources constraints appear.
  - **Could have:** These features enhance the system with greater functionality, but the timeliness of their delivery is not critical.
  - **Want to have:** These features serve only a limited group of users and do not drive the same amount of business value as the preceding items.

Kano Model

- The Kano model was created to help companies learn how to build better products to “delight” the customers.

- The model assists with the prioritization process because it breaks down features into three categories.

Kano Model

- The first is **basic needs**, which are sometimes referred to as satisfiers/dissatisfiers. These are the features that must be there for the product to work, in the eyes of the customer: The functionality is expected, and if it is not included, it causes great dissatisfaction.

- After a consumer's basic needs are met, then there are **performance needs**, which add necessary functionality to enhance the user experience.

Kano Model

- The final category, which results in true product differentiation, consists of features that <u>**delight**</u> or excite users, and they are an unexpected bonus from the consumer's viewpoint.

- The challenge when it comes to prioritization is to focus on enough of the basic needs, performance needs, and delighters to deliver a compelling product to the marketplace.

Kano Model

- If you have a product with only delighters but no basic needs, then you might have something really innovative that does not work.
- Conversely, if you have something that satisfies every basic need but does not delight, then customers may not be compelled to buy it because it does not offer anything special or different from the competition.

Kano Model

- Another aspect of the Kano model that is critical to understand is the evolution of the feature categorization: Over time, delighters will become performance needs, and performance needs will become basic needs.

- Therefore, companies must continue to prioritize delighter features if they want to remain innovative.

Estimation

Estimation

- In addition to effective prioritization, grooming and planning sessions require sound estimates for how long the work is likely to take.

- Estimation is one of the hardest things to do in software development whether using an agile methodology or not.

Estimation

- Indeed we cannot predict the future and estimates will always vary and change throughout a project.

- There are a number of different ways to come up with the estimates, just as there are many different ways to prioritize the feature requests.

Level of Effort (LOE) or T-Shirt Sizing

- Perhaps the least precise but easiest method of estimating goes by one of the following names: “T-shirt size,” “level of effort” (LOE), or “small, medium, large.”

- The last title is the most descriptive because the development team just estimates each story as small, medium, and large; teams may add extra-small and extra-large, depending on what they have decided as their measure.

Level of Effort (LOE) or T-Shirt Sizing

- The advantages of this approach are that it is very simple and can be done quickly.
- The disadvantages include a lack of precision and an inability to add up several stories into a meaningful measure.
- For example, it is hard to say “our team can work on two smalls and one medium at a time.” For these reasons, teams often use this form of estimating early in the process and refine it later.

Ideal Time (Days or Hours)

- In this form of estimating, the developer determines the amount of time that a task would take under ideal circumstances—meaning no interruptions, no meetings, and no phone calls—and that is set as the ideal time.

- This is an easier metric for developers to assign than actual clock time because it is difficult for them to factor in all of the possible interruptions that might occur.

Ideal Time (Days or Hours)

- Each organization will have some factor that is applied to convert ideal time to clock time; a ratio of 2 to 1 is fairly common, meaning that within every hour, 40 minutes are spent on concentrated work, with the other 20 minutes consumed by interruptions or breaks.
- Ideal time is a useful metric because it is relatively easy for a developer to estimate, provided that the requirements/user stories are well defined and thorough.

Hours

- “Hours” are simply the number of hours that the development team estimates will be required to complete the user story.

- This is an accurate way to measure because there is only one definition for an hour—60 minutes—so there is no room for misinterpretation.

Hours

- The disadvantage to using hours at this stage (early planning) is that the team might be lacking information to make such a precise estimate.

- Because hours are so defined, there is a greater risk of executives and stakeholders placing inappropriate expectations on the team to deliver exactly in the estimated time frame.

Story Points

- In Scrum, many teams use the concept of “story points,” which are an arbitrary measure that allow the teams to understand the size of the effort without the potentially binding expectations that come with hours.
- When estimating in story points, many teams use the Fibonacci sequence, a mathematical tool defined in the thirteenth century for rational approximations.

Story Points - Fibonacci Sequence

- The sequence is the sum of the prior two numbers: 1, 2, 3, 5, 8, 13, 21; as you can see, 1+2=3, 2+3=5, 3+5=8, and so on.

- The Fibonacci sequence is helpful for Agile estimating because as the points get higher, the degree of uncertainty is increasing.

Story Points - Fibonacci Sequence

- For the purposes of Agile estimating, the sequence has been modified to account for the needs of software development.
- Some teams have added a ½ for the user stories that are so small in effort they really do not need estimating.
- Also, many teams replace the 21 with 20 and add 40 and 100; the idea is that if a story is estimated at greater than 13 story points, there is a good chance that it is still an epic and needs to be broken down.
- The difference between a 20, 40, or 100 is just a sense for the size of the epic.

Story Points - Fibonacci Sequence

- Secondly, some teams add 200, 500, and/or infinity (∞). The reason for these additions is that people (typically management or stakeholders) are used to comparing things as though they relate, which is not always the case.

- For example, the team estimates an activity at 100 points because it is an epic and they do not have enough information to make a more precise assessment. As that feature is better understood and broken down, it is eventually delivered in five months.

Story Points - Fibonacci Sequence

- Some members of the management team might now equate a 100-point story with a five-month delivery cycle, which is not true at all.

- The next 100-point story—which is equally epic and lacking in information—may take twice as long to deliver, so there is no relative comparison. To eliminate the opportunity to compare, some teams have started using infinity to represent "not enough information."

Story Points - Fibonacci Sequence

- One additional thing to keep in mind is that unlike hours, which have a fixed and commonly understood definition, story points are more abstract and negotiable.

- One team may assign 3 points to a story where that same story on another team would have been a 2 or a 5.

- As long as everyone on the team agrees to the size of a story point, then all of the estimating will be coherent.

Story Points - Fibonacci Sequence

- There are a couple of other considerations on either side of the spectrum.
- First, some teams add a 0 (zero) to their Fibonacci sequence, but one might question why a story with zero points or no level of effort is even in the mix.
- Some teams use this as a placeholder to remind them of an activity that another team must complete; a dependency).
- For example, this task requires no effort on the part of Scrum team A but needs to be delivered by Scrum team B before Scrum team A can complete its work; therefore, it sits in Scrum team A’s backlog with zero points.

Story Points - Fibonacci Sequence

- The easiest way for a team to get started using story points is through relative sizing; this involves identifying a task that everyone is familiar with and assigning a point value to it.

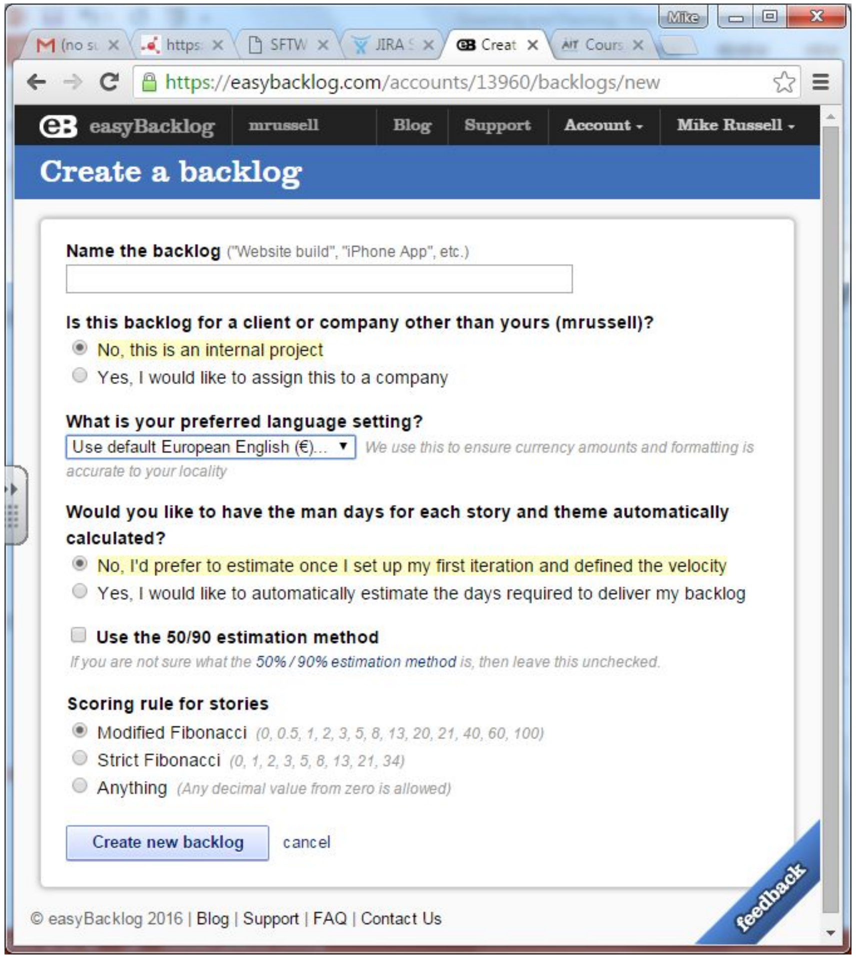
- Let's say that adding a field to the database happens frequently, so everyone is aware of the effort and time required.

- The team assigns that activity as 5 story points, and then every subsequent story is compared to that one. Is this activity bigger or smaller than adding a field to the database? If it is bigger, then it could be an 8 or a 13; if it is smaller, it could be a 2 or a 3. Starting with something relative provides a benchmark for new teams.

Story Points - Fibonacci Sequence

- It is also worth noting that the testing activities should be included in the story point estimation. If an action is easy to develop but challenging to test, then it should receive a higher estimate because designing, coding, and testing are all part of completing a story.

Story Points - Fibonacci Sequence: Easybacklog.com



Create a backlog

Name the backlog ("Website build", "iPhone App", etc.)

Is this backlog for a client or company other than yours (mrussell)?

- [✓ No, this is an internal project]
- Yes, I would like to assign this to a company

What is your preferred language setting?

Use default European English (€)... *We use this to ensure currency amounts and formatting is accurate to your locality*

Would you like to have the man days for each story and theme automatically calculated?

- [✓ No, I'd prefer to estimate once I set up my first iteration and defined the velocity]
- Yes, I would like to automatically estimate the days required to deliver my backlog

☐ **Use the 50/90 estimation method**

If you are not sure what the 50% / 90% estimation method is, then leave this unchecked.

Scoring rule for stories

- [✓ Modified Fibonacci *(0, 0.5, 1, 2, 3, 5, 8, 13, 20, 21, 40, 60, 100)*]
- Strict Fibonacci *(0, 1, 2, 3, 5, 8, 13, 21, 34)*
- Anything *(Any decimal value from zero is allowed)*

Create new backlog cancel

© easyBacklog 2016 | Blog | Support | FAQ | Contact Us

50/90 Estimation: Easybacklog.com

- In some environments developers have to come out with fixed costs quotes for their customers.
- In such situations, most people estimate bottom up and top down with their experience and then tend to slap on a ‘buffer’ or contingency as they call it.
- This is a widely used technique but unfair on your customer, you are basically adding a buffer on all requirements or stories regardless of complexity or detail that you have at the time.

50/90 Estimation: Easybacklog.com

- Instead Easybacklog.com use the 50/90 estimation technique which asks developers to estimate a story with 50% margin (they will be right 50% of the time) and one with a 10% margin (they will be right 90% of the time).

- We type that in into Easybacklog.com for each story and it calculates a buffer for the backlog using the square root of the sums of the squared differences between the 50 and 90 estimates.

- Still following me? Don’t worry Easybacklog.com calculates it all for you at the story, theme and backlog level.

- This gives you a better and fairer buffer on your estimates and costs as it will buffer where it needs it.

Estimation: Team Participation

- How does an Agile team actually execute on the estimation process?

- In all instances, we are looking for participation from the developers and testers because we want to benefit from the wisdom of self-organizing teams. Outlined next are several strategies that teams have used to come to consensus on estimation.

Estimation: Team Participation – Planning Poker

- Having the development team participate in the activity of estimating can have distinct advantages over asking a single developer to size a feature.

- If the team is using story points one of the fun and effective ways to increase engagement is through a game called “planning poker.”

- Each team member is given a set of number cards with the modified Fibonacci sequence—1, 2, 3, 5, 8, 13, 20, 40, 100.

Estimation: Team Participation – Planning Poker

- After the product owner and the team have discussed the user story in detail and everyone believes they have an adequate understanding, the Scrum master or coach will call for each team member to lay down the card representing the number of story points that he or she assigns to this user story.

- Many Scrum masters ask all team members to “throw” at the same time so that no one can be influenced by their teammates.

Estimation: Team Participation – Planning Poker

- In addition to an estimated size, we are looking to see if there is a discrepancy.

- If a user story is discussed and one team member throws a 2 but another throws a 13, then there is a difference that needs to be discussed.

- It does not mean that one person is right and the other is wrong; this actually identifies a lack of common understanding as it relates to the user story.

Estimation: Team Participation – Planning Poker

- Just like all of the other opportunities within Agile for collaboration and discussion, this discrepancy should be explored.

- Why did the developer who threw the 13 think that this task was so big? Is there a point of interconnection that the other developer (who threw the 2) was not thinking of? Did they just understand the request differently? Or has the low estimator done something like this before, so he or she is very confident in that assessment?

Estimation: Team Participation – Planning Poker

- Again, the value is in the dialogue, so everyone can come to a common understanding of the request and what it will take to fulfill it.

- The result could be that the team agrees on a 2 or a 13 or on a number in between. The point of the exercise is to facilitate collaboration and negotiation so the team is clear on the desired feature and confident in their ability to deliver.
- If you have a distributed team so not all team members are in one location, there are tools such as http://www.planningpoker.com/ that allow you to have the same poker game over the Internet.

Estimation: Team Participation – Wide-Band Delphi

- The Delphi method takes the same approach as planning poker but addresses the estimating in a more structured and formal way.
- A facilitator calls a meeting of the experts and a feature request is discussed.
- Each expert participant then completes an anonymous estimation form and submits it to the facilitator.
- The facilitator compiles the responses and redistributes them to the experts—continuing to maintain the anonymity of the respondents.

Estimation: Team Participation – Wide-Band Delphi

- Based on the new inputs, the expert team discusses the feature request again, clarifies any incorrect assumptions, gains a better understanding of the feature, and submits new estimation forms to the facilitator.

- This process is repeated until the estimations are an agreed-upon value.

Estimation: Team Participation – Wide-Band Delphi

- Although there is more effort to the Delphi method, it may produce more accurate results because of the anonymity of the respondents, which prevents a strong personality from swaying the group.

- The disadvantages are that it is time-consuming, and the experts are not required to reveal their biases to the group.

Estimation: Team Participation – Extreme Programming (XP)

- Within XP, it is critical that the developers are the ones making the estimates and no one else; they are also the only ones who can make changes to estimates.

- The goal is to instill a degree of responsibility or ownership of the estimates to the developers.

- The XP process also provides a feedback loop for the actual time required, so developers can continue to improve their estimates over time.

Feature-Driven Development and Kanban

Feature-Driven Development

- Feature-driven development (FDD) takes a different approach to the concept of requirements.
- FDD breaks down the development team differently and does not adopt the idea of collective development, where a single team is working together in the code to deliver a feature.
- FDD works with the concept of “classes” that must be integrated to deliver the full feature set.
- To manage requirements, FDD breaks the features down into a hierarchy of three levels.

Feature-Driven Development

- The highest level is the problem domain or the subject area, which is then broken down into the business activities, and then the individual features are assigned to a business activity.
- This is a more model-driven approach, referred to as the “plan by feature” process, and allows FDD to support large projects and teams.

Kanban

- Kanban follows a different practice when it comes to planning and executing on an Agile work effort.

- We will discuss in the section on Tracking and Reporting.

Conclusion

Conclusion

- Grooming and planning are critical activities within the Agile practice because they incorporate the values and principles that we aspire to regarding self-organizing teams, face-to-face collaboration, sustainable work environments, and frequent delivery of working software.
- If teams do not spend adequate time with backlog grooming and focus on ensuring that user stories/requirements are detailed appropriately, prioritized, and estimated, it is extremely difficult for the developers and testers to execute on their work.

Conclusion

- Likewise, if the sprint planning efforts are sloppy or undervalued, then the importance of delivering on the committed work is marginalized and teams will not excel.

- Agile provides a structured framework to facilitate important conversations about difficult work; through this framework, teams can elevate their performance and deliver real business value to customers and the organization.

Scrum: Product Backlog Grooming

Product Backlog Grooming

- The activities of prioritization and estimation often take place in a session called *product backlog grooming*.

- This is typically a meeting between the product owner and the Scrum master/coach and may include the entire development team.

Product Backlog Grooming

- The grooming session is also used to clarify and improve on the user stories; this could include identifying and breaking down requirements that are too big (epics), improving poorly written stories, and adding acceptance criteria.

- Once the participants have a clear understanding of the features/stories, they can discuss the prioritization of those stories and add the estimates.

Product Backlog Grooming

- The goal of the grooming session is to leave the meeting with good, solid user stories that are prioritized, discussed, well understood, negotiated, and agreed upon to ensure a successful sprint.

- The product owner is responsible for leading the grooming session.

Product Backlog Grooming

- Good product owners value the dialogue between the participants and recognize the benefits of the diverse backgrounds and perspectives of the group.

Scrum: Sprint Planning

Additional Inputs – Sprint Planning

- Once the backlog grooming is complete and the team understands the highest priority user stories in sufficient detail, the next meeting in the Scrum methodology is Sprint planning.

- Before we can effectively execute on that meeting, several additional concepts/data points must be understood/collected.

Additional Inputs – Sprint Planning: Team Velocity

- Within Agile, as teams spend time together and work through a few sprints, they start to get a sense for their “velocity”: This is the amount of work that the team can usually deliver within the time frame of a sprint or iteration and can be used as a predictor for future iterations.

- Story points are the most common units of measure for velocity, though hours could be used if the team was estimating in hours.

Additional Inputs – Sprint Planning: Team Velocity

- It would be challenging to calculate velocity for a team estimating in t-shirt sizes (or small/medium/large) because the estimates cannot be mathematically summed.

- When a team first comes together, it is difficult to estimate their velocity because they have not yet experienced an iteration as a team.

Additional Inputs – Sprint Planning: Team Velocity

- The first few iterations are usually more guesswork than precision, and that is fine because it provides the opportunity to inspect and adapt.

- Once the team is established and finds their rhythm, velocity is a great tool to use when establishing their commitment for future iterations.

Additional Inputs – Sprint Planning: Team Velocity

- Nevertheless, a few factors can disrupt velocity.
- If the team members are not dedicated to the team full-time, for example, it is hard to predict their availability to work on the next iteration.
- Similarly, if the team members change frequently, it is difficult for the team to find their rhythm

Additional Inputs – Sprint Planning: Team Velocity

- Another negative impact to velocity is modifying the length of the sprint: If the team works on a two-week sprint, then a one-week sprint, then a two-week sprint again, it becomes difficult to gauge their capacity over a specified time period.

- Ideally, you want a team to work together for several iterations of consistent duration; then you will have the data you need to calculate the team’s velocity for planning purposes.

Additional Inputs – Sprint Planning: Team Velocity

- Personnel considerations for things such as vacation and training are yet another dynamic that will affect velocity.

- For example, if a team will have two members on holidayfor half of their iteration, then clearly they should not commit to the same level of work as if everyone were present

Additional Inputs – Sprint Planning: Team Velocity

- Other elements that affect velocity could be external to the team, such as the stability of the infrastructure.
- If the development team is continually called upon to fix production problems, then they will have less time to spend writing new code.

Additional Inputs – Sprint Planning: Team Velocity

- There is no defined number of sprints that must be completed in order to calculate velocity; it is an ongoing, evolving measurement to be used as an input for the planning process.
- Velocity should *not* be used as a management metric since it is based on story points, which are an arbitrary unit of measure defined by each individual team; teams with a higher velocity may not be more productive than others with lower velocity.

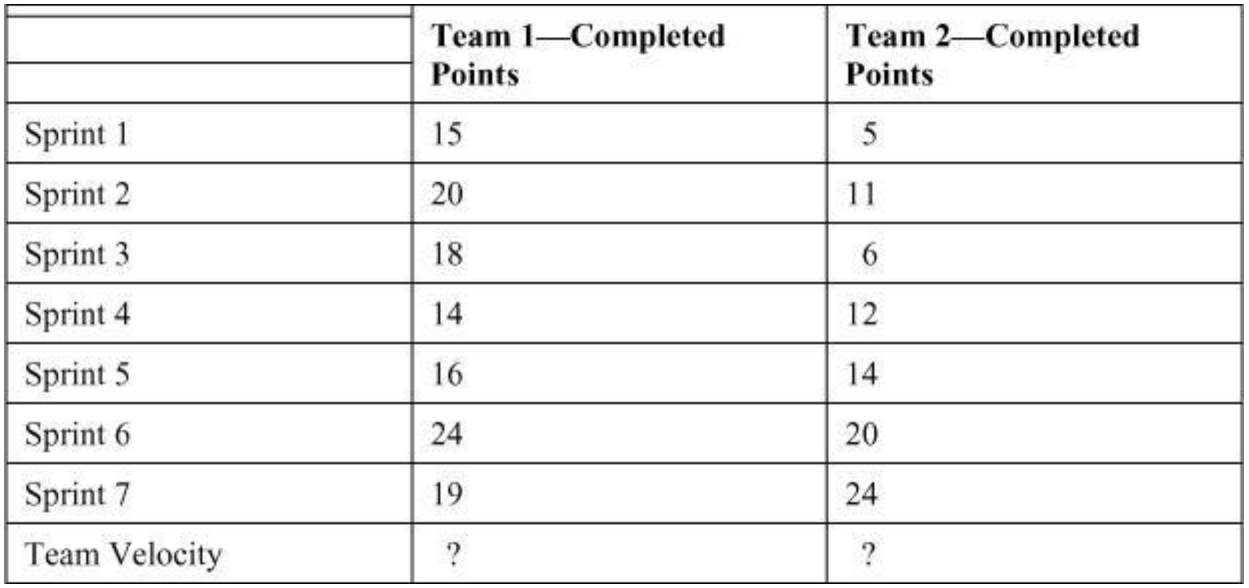
Additional Inputs – Sprint Planning: Team Velocity

- In fact, management needs to be careful to avoid using velocity as an indicator of performance or productivity because it could lead to unhealthy behaviors on the team

Additional Inputs – Sprint Planning: Team Velocity

- There are a few different ways to calculate velocity over time.
- Some teams simply average their totals for all of the sprints/iterations that they have worked on.
- Other teams, particularly those that are new to Scrum and are improving in their abilities, will use the concept of "*yesterday's weather*," therefore using only the last two to three sprints as indicators of their future capacity.

Additional Inputs – Sprint Planning: Team Velocity



	Team 1—Completed Points	Team 2—Completed Points
Sprint 1	15	5
Sprint 2	20	11
Sprint 3	18	6
Sprint 4	14	12
Sprint 5	16	14
Sprint 6	24	20
Sprint 7	19	24
Team Velocity	?	?

Exercise:

Calculate the velocity of team 1 using averages for all data points. Calculate the velocity of team 2 using "yesterday's weather" by including only the last two sprints. Which team has higher productivity? What are some of the reasons that velocity on team 1 might drop to 10 on the next sprint?

Additional Inputs – Sprint Planning: Team Velocity

- Extreme Programming also has the concept of velocity for planning purposes, but it allows for two types of planning:
  - You can either set a date and determine how many stories can be implemented by that date, or define a specific scope and determine how long it will take to finish a defined set of stories.

- Either way, the team’s velocity is used to make the projections

Additional Inputs – Sprint Planning: Definition of "Done"

- Within several of the Agile methodologies, including Scrum, there is an element called the **definition of "done**."
- This is a meaningful conversation that the development teams, product owner, and stakeholders need to have to ensure a common understanding of completion.
- When different people have a different definition of done, expectations will not be met and dissatisfaction can result.

Additional Inputs – Sprint Planning: Definition of “Done”

- Does your team include code reviews in your definition of done?
- Does your product owner think that the team is going to update the release notes before a sprint is considered completed?
- These are important conversations to have to ensure that everyone is on the same page.

Additional Inputs – Sprint Planning: Definition of “Done”

- Every new team should dedicate time to discussing and agreeing on their definition of done, which may or may not include elements such as code that is checked in and integrated and has automated tests.
- When the definition of done is clear and well understood, then we ensure that the conclusion of every sprint will meet everyone’s expectations.

Additional Inputs – Sprint Planning: Definition of “Done”

- Extreme Programming handles the definition of done slightly differently: When a task is completed, it is crossed off of the board.
- To ensure that done means done, in XP, only the customer (not the developer) can cross off a task.
- Remember that the customer in XP is equivalent to the product owner in Scrum.

Additional Inputs – Sprint Planning: Incorporation of Technical Debt

- One of the discussion points with regard to the product backlog is the inclusion of technical debt.

- With Agile, teams are moving very quickly, and sometimes to achieve the sprint goal within the time frame allotted, the team, either knowingly or unknowingly, creates technical debt.

Additional Inputs – Sprint Planning: Incorporation of Technical Debt

- Technical Debt includes those internal things that you choose not to do now, but which will impede future development if left undone. This includes deferred refactoring.

- There are two basic kinds of debt—unintended and intentional.

Additional Inputs – Sprint Planning: Incorporation of Technical Debt

- Unintended technical debt is simply the consequences of the team’s learning curve.
- Perhaps they designed something poorly or did not test thoroughly enough, or the business was not 100% certain about the business requirements.
- No matter what the root cause, this type of debt is unintentional—no one meant to write bad code or be sloppy—and it happens as part of the continuous learning process that is essential to Agile.

Additional Inputs – Sprint Planning: Incorporation of Technical Debt

- Intentional technical debt is incurred as trade-offs are made in the development process.

- We may choose to incur technical debt for short-term, tactical reasons or long-term, strategic ones.

- For example, creation of an un-scalable feature for the initial release, knowing that at some late point the feature will need to be re-done to its scalability requirements.

Additional Inputs – Sprint Planning: Incorporation of Technical Debt

- When we are ready to tackle technical debt, should it be included in the backlog of prioritized stories?
- Although there is some debate on this topic, most experts agree that it should.
- The technical teams may have to argue their case to the product owner to prioritize the efforts appropriately, and a good product owner should listen and understand the impact of allowing the debt to fester.

Additional Inputs – Sprint Planning: Bugs

- Bugs, also referred to as defects, are different from technical debt in that they are usually errors in the writing or implementation of the code that are impeding the performance or usability of the application.
- In many companies, bugs are handled outside of the feature release (sprint) process because of their immediacy.
- Within organizations where bug fixes are resource intensive and can wait for the development cycle, bugs should also be included in the backlog and prioritized accordingly.

Scrum: Sprint Planning

- In Scrum, once we have the necessary inputs, we are ready for the Sprint planning meeting.

- In this meeting, we are going to decide exactly what we are going to work on during the sprint in order to plan for it.

Scrum: Sprint Planning - Inputs

- There are a number of inputs to the Sprint planning meeting, such as the following:
  - Groomed and prioritized backlog
  - Estimates for the highest priority stories
  - The velocity of the team
  - Definition of done
  - The schedule of the sprint—are there vacation days or a company holiday that will affect the amount of work we can commit to?
  - Other inputs—are team members getting new monitors, is there an All-Hands meeting that everyone is required to attend?

Scrum: Sprint Planning – The Planning Session/Meeting

- The sprint planning meeting has two distinct objectives.
- First, we want to make sure that everyone on the team has a complete understanding of the user stories.
- Team members may not all have attended the backlog grooming session, so it is important that everyone have the opportunity to ask questions and gain a clear understanding of the top priority stories.

Scrum: Sprint Planning – The Planning Session/Meeting

- The second objective is to allow the team to break the work down into individual tasks and determine who will own each piece of work.

Scrum: Sprint Planning – The Planning Session/Meeting

- The planning process is as follows:
  - **1.** Once we understand the user stories, the team will decide how many stories can be committed to during this sprint. This is the sprint goal.
  - **2.** The team then breaks the user stories down into individual tasks.
  - **3.** Each team member selects the tasks that he or she is willing to own and complete.
  - **4.** The team member then estimates—in hours—how long it will take to complete each task.

Scrum: Sprint Planning – The Planning Session/Meeting

- With item #2, user stories are broken down into tasks.
- Some teams skip this step and a single developer is responsible for the entire user story delivery.
- More frequently, each story will contain several distinct pieces—for example, there is an update to the user interface, a new database field, an update required for the API to add the new field, and so on. Each of those individual tasks can be owned by different developers depending on their level of expertise.

Scrum: Sprint Planning – The Planning Session/Meeting

- In item #3, team members select the tasks that they will own, which creates an opportunity to cross-train or develop new skills.
- In step #4, the team might add some time to their estimates of how long it will take to complete some of the identified tasks.

Scrum: Sprint Planning – The Planning Session/Meeting

- Another important outcome of the Sprint planning meeting is that story points are converted to actual hours and the hours are attributed to the tasks, not the stories.

- The thought is that story points are appropriate for stories in backlog grooming and hours are appropriate for tasks in Sprint planning because the level of detail is increasing and therefore the confidence in estimating is greater and can be more precise.

Scrum: Sprint Planning – The Planning Session/Meeting

- For the teams that estimate in hours to begin with, this step is still important, because the original estimates may need to be modified now that the team knows more and the developer who will be doing the work is known.

Scrum: Sprint Planning – Output

- There are two primary outputs from the Sprint planning meeting.
- First is the Sprint goal, the user stories that are committed to for this sprint.
- Second is the Sprint backlog, which is the list of tasks and owners. The Sprint backlog is the team's "to do" list for the sprint.

Scrum: Sprint Planning – Chickens and Pigs

- The Sprint planning meeting also demonstrates the role assignment for chickens and pigs.
- In the first half of the Sprint planning meeting, when the user stories are being described and discussed, the product owner is absolutely a pig.
- He or she is deeply invested in the conversation and is committed to the positive outcome of the meeting.

Scrum: Sprint Planning – Chickens and Pigs

- When the meeting shifts to the developers breaking down the stories into tasks and selecting their tasks, the product owner’s role shifts to that of a chicken.

- He or she is interested in the outcome of the conversation but the level of commitment has been reduced, since he or she will not be doing the actual work (coding and testing.)

Extreme Programming: XP Planning Game

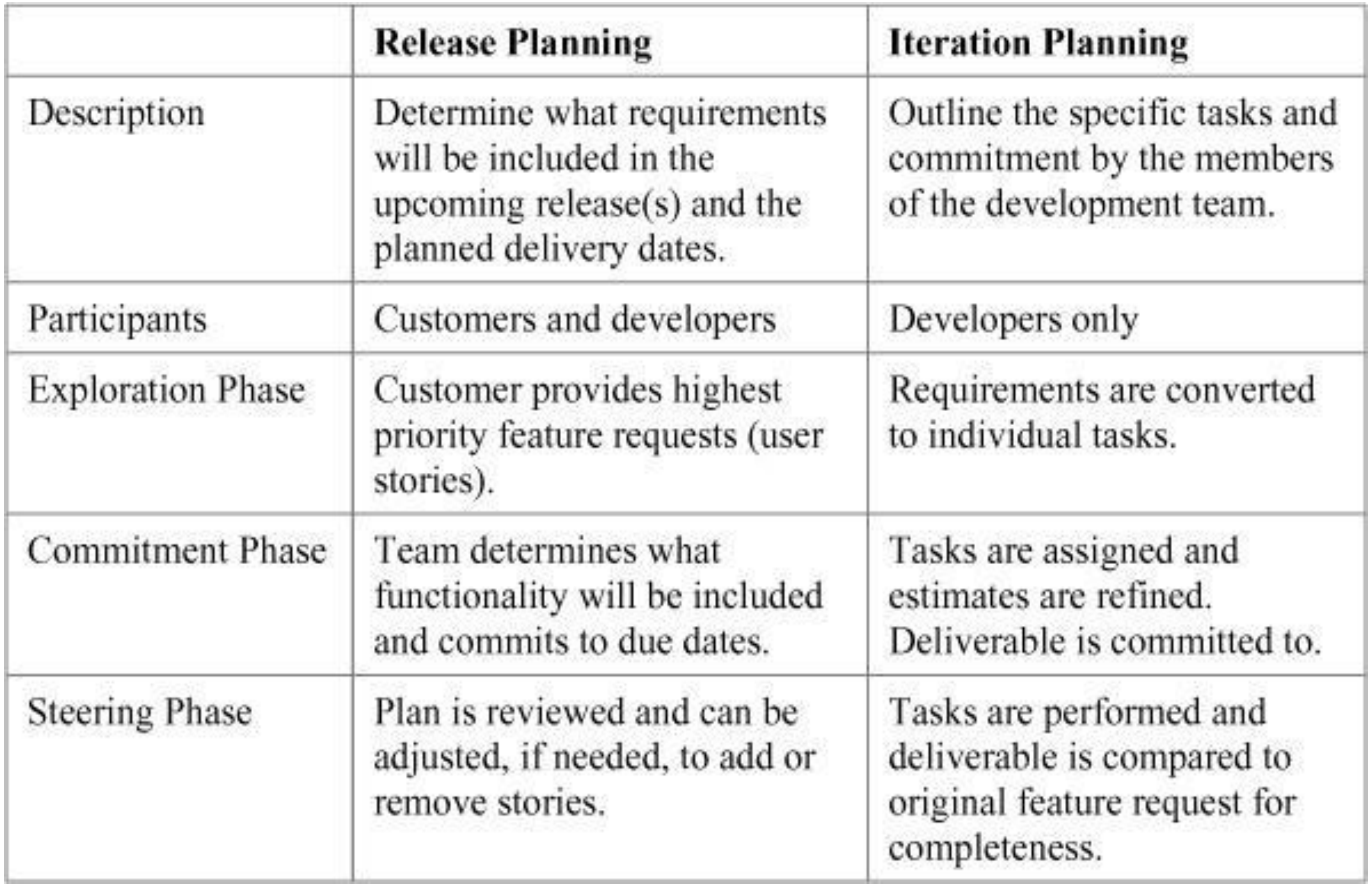
XP Planning Game

- XP has a ritual similar to the Scrum backlog grooming session and the Sprint planning meeting: the planning game.

- The planning game also consists of two parts: release planning, which is most similar to backlog grooming, and iteration planning, which is most similar to Sprint planning.
- Each meeting has three distinct phases—exploration, commitment, and steering.

- The table on the next slide compares the two aspects of the planning game side by side.

XP Planning Game



	Release Planning	**Iteration Planning**
Description	Determine what requirements will be included in the upcoming release(s) and the planned delivery dates.	Outline the specific tasks and commitment by the members of the development team.
Participants	Customers and developers	Developers only
Exploration Phase	Customer provides highest priority feature requests (user stories).	Requirements are converted to individual tasks.
Commitment Phase	Team determines what functionality will be included and commits to due dates.	Tasks are assigned and estimates are refined. Deliverable is committed to.
Steering Phase	Plan is reviewed and can be adjusted, if needed, to add or remove stories.	Tasks are performed and deliverable is compared to original feature request for completeness.

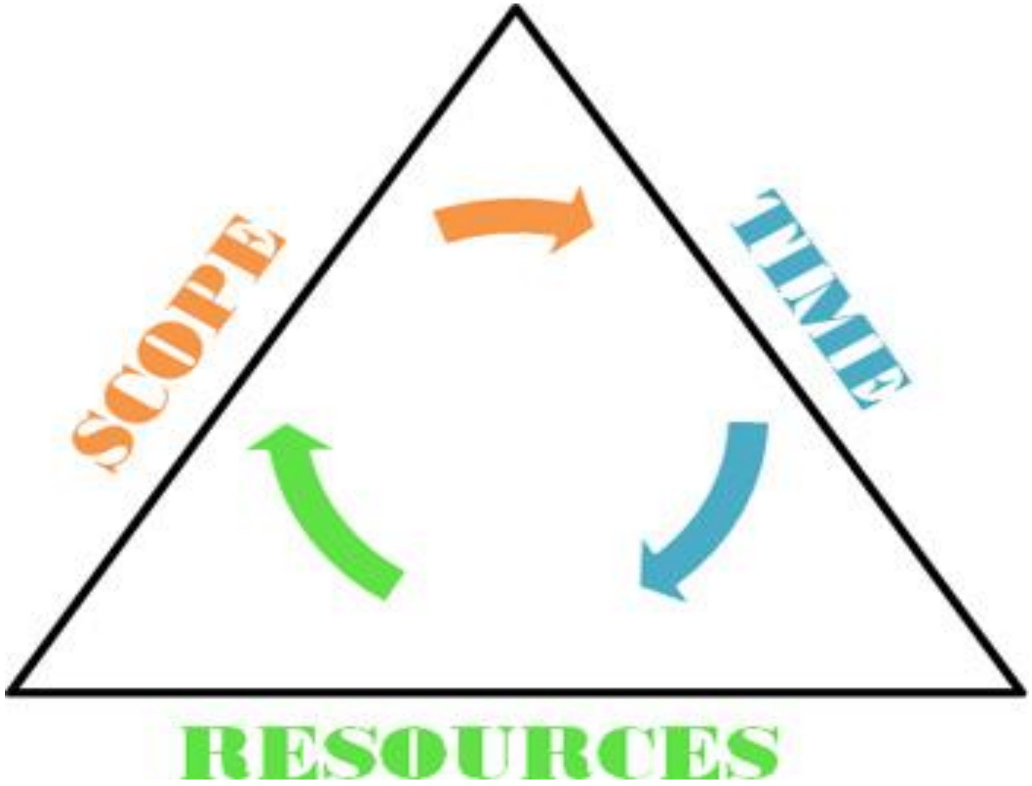
XP Planning Game

- It is important to note the participants in the different parts of the planning game.
- Because customers have the most information about value—they’re most qualified to say what is important—they *prioritize*.
- Because programmers have the most information about costs—they’re most qualified to say how long it will take to implement a story—they *estimate*.

- Neither group creates the plan unilaterally. Instead, both groups come together, each with their areas of expertise, and play the planning game.

Project Management

Concept of Triple Constraints



Concept of Triple Constraints

- There is a long-held principle within the project management discipline that projects are made up of three variables: schedule/time, cost/resources, and scope/quality (see previous slide).
- The theory is that you cannot adjust one without affecting the other two.
- If you shorten the schedule and say that a release is due one month earlier than planned, then you will also need to add resources or reduce the scope in order to meet the new desired date.

Concept of Triple Constraints

- Likewise, if you remove resources from a project, you will need to either extend the time frame to push out the delivery date or reduce the scope.
- Finally, if you add to the scope, then you will need to either extend the time frame or add resources.
- This model certainly helps with project management because it clearly depicts the trade-offs that must be made when a significant variable changes.

Agile and Concept of Triple Constraints

- Agile looks at the theory of triple constraints slightly differently.

- First, there is a school of thought that scope and quality are, in fact, two very different things:
  - A team could deliver the prescribed scope at lesser quality, or it could maintain acceptable quality standards with a reduced scope.

Agile and Concept of Triple Constraints

- Planning a project may then be quantified by four variables; scope, resources, time, and quality.
- Scope is how much is to be done.
- Resources are how many people are available.
- Time is when the project or release will be done.
- And quality is how good the software will be and how well tested it will be

Agile and
Concept of Triple Constraints

- No one can control all [four] variables. When you change one, you inadvertently cause another to change in response.

- It is important to understand the concept of the triple constraints when engaged in planning and grooming, because one must grasp that changing one variable will impact other variables and appropriate adjustments will be needed.

Agile and Concept of Triple Constraints

- Another way to look at this in an Agile environment is to suggest that resources and time are fixed.
- The team size is five to nine people, so that cannot change. The date is fixed by the duration of the sprint, in many cases two weeks.
- Therefore, scope must be flexible to be able to be designed, coded, and tested within the confines of the sprint.

Agile and Concept of Triple Constraints

- By managing the size and priority of the stories in the backlog, you are effectively managing the scope while the time and resources are fixed.

Test Case Design and Coverage Techniques

References

- Standard for Software Component Testing
  - ©British Computer Society, SIGIST, 2001
    Working Draft 3.4 (27-Apr-01)

Techniques and Measures (Coverage)

- The Standard defines test case design techniques and test measurement (coverage) techniques.

- The techniques are defined to help users to
  - design test cases
  - and to quantify the testing performed (coverage).

- The definition of test case design techniques and measures provides for common understanding in both the specification and comparison of software testing.

Test Case Design and Measurement (Coverage) Techniques

- Black Box
  - Equivalence Partitioning
  - Boundary Value Analysis
  - Classification Trees

- White Box
  - Statement Coverage
  - Branch Coverage

Test Case Design and Measurement (Coverage) Techniques

- Each technique has its own “identified” coverage items.
- Coverage measurements determine which coverage items have been tested and which have not been tested.
- Usual to try and achieve 100% coverage for each technique.

Equivalence Partitioning – Design

- Equivalence partitioning is based on the premise that the inputs and outputs of a component can be partitioned into classes that, according to the component's specification, will be treated similarly by the component.
- Thus the result of testing a single value from an equivalence partition is considered representative of the complete partition.
- Both valid and invalid values are partitioned in this way.

Equivalence Partitioning

Black Box Technique

Equivalence Partitioning – Design

- Test cases shall be designed to exercise partitions.
- A test case may exercise any number of partitions. A test case shall comprise the following:
  - the input(s) to the component;
  - the partitions exercised;
  - the expected outcome of the test case.
- Test cases are designed to exercise partitions of valid values, and invalid input values.
- Test cases may also be designed to test that invalid output values cannot be induced.

Equivalence Partitioning – Coverage

- Coverage items are the partitions identified by the model.
- Coverage is calculated as follows:

$$\text{Equivalence partition coverage} = \frac{(\text{number of covered partitions})}{(\text{total number of partitions})} * 100\%$$

Equivalence Partitioning – Example

- Read the requirements.
- Divide the range of possible inputs (and outputs) into partitions / classes.
- For all values within a class, the software is expected to do “the same thing”.
- We test one value (at least) from each class.

Equivalence Partitioning – Example

- The valid range for the month is 1 to 12, representing January to December.
- This valid range is called a partition.
- In this example there are two further partitions of invalid ranges. The first invalid partition would be <= 0 and the second invalid partition would be >= 13.

Equivalence Partitioning – Example

```
  ... -2 -1 0   1 ............. 12   13 14 15 .....
--------------|-------------------|---------------------
invalid partition 1     valid partition          invalid partition 2
```

Equivalence Partitioning – Example

- Select one value from each partition to test the behaviour of the program. To use more or even all values of a partition will not find new faults in the program.
- The values within one partition are considered to be "equivalent". Thus the number of test cases can be reduced considerably.
- An additional effect of applying this technique is that you also find the so called "dirty" test cases. An inexperienced tester may be tempted to use as test cases the input data 1 to 12 for the month and forget to select some out of the invalid partitions. This would lead to a huge number of unnecessary test cases on the one hand, and a lack of test cases for the dirty ranges on the other hand.

Identifying equivalence partitions

Validity Rule	*Input Requirement*	*Valid Equivalence Class(es)*	*Invalid Equivalence Classes*
Range	Between 0.0 and 99.9	Between 0.0 and 99.9	<0.0 >99.9
Count	Must be 1, 11, 43, 82, 955	1, 11, 43, 82, 955	any number not in the valid list e.g. 2, 37, -56
Set	Must be BUS, CAR or TAXI	BUS CAR TAXI	e.g. XXX, TRAIN, PLANE
Must	Must begin with a letter	Begins with a letter (a-z, A-Z)	Any non alpha character or null e.g. 6, +, >, ~

Output partitions

- Output equivalence partitions are also important:
  - calculate tax brackets - can use different input salary ranges

Equivalence Partition Coverage

$$\text{Equivalence partition coverage} = \frac{\text{number of covered partitions}}{\text{total number of partitions}} * 100\%$$

Boundary Value Analysis

Black Box Technique

Boundary value analysis

- BVA is technique in which tests are designed to include representatives of boundary values.

- Experience shows more faults at boundaries of equivalence partitions where partitions are continuous.

- The maximum and minimum values of a partition are its boundary values.

- Don't forget test cases for output partitions

Boundary value analysis

- The expected input and output values should be extracted from the component specification.

- The input and output values to the software component are then grouped into sets with identifiable boundaries (using Equivalence Partitioning).
- Each set, or partition, contains values that are expected to be processed by the component in the same way.

Boundary Value Coverage

$$\text{Boundary value coverage} = \frac{\text{number of distinct boundary values executed}}{\text{total number of boundary values}} * 100\%$$

Basic Test Template

Test Template

- **Test Identifier** - Unique number to identify test.
- **Test Objective** - Objective of test.
- **Test Input(s)** - Input values to drive the test.
- **Expected Output(s)** - Expected Output(s) to be generated if the test is a success.

Classification Trees

Boundary Value Analysis – Example

```
~ … -2 -1 0   1 ………….. 12   13 14 15 ….. ~
--------------|-------------------|---------------------
invalid partition 1      valid partition          invalid partition 2
```

- What about the boundaries at ~ ?

What is the Classification Tree Method (CTM)?

- Testing is a step in the software development process. However, the planning of such testing often raises the same questions:
  - How many tests should be run?
  - What test data should be used?
  - How can error sensitive tests be created?
  - How can redundant tests be avoided?
  - Have any test cases been overlooked?
  - When is it safe to end testing?
- The Classification Tree Method offers a systematic procedure to create test case specifications based on a problem definition.

Using Classification Trees

- Black box test “technique” aimed at preparing test cases.
- Appropriate where functionality depends on combinations of conditions.

- CTs useful in any ‘combinatorial’ situation.

An Example

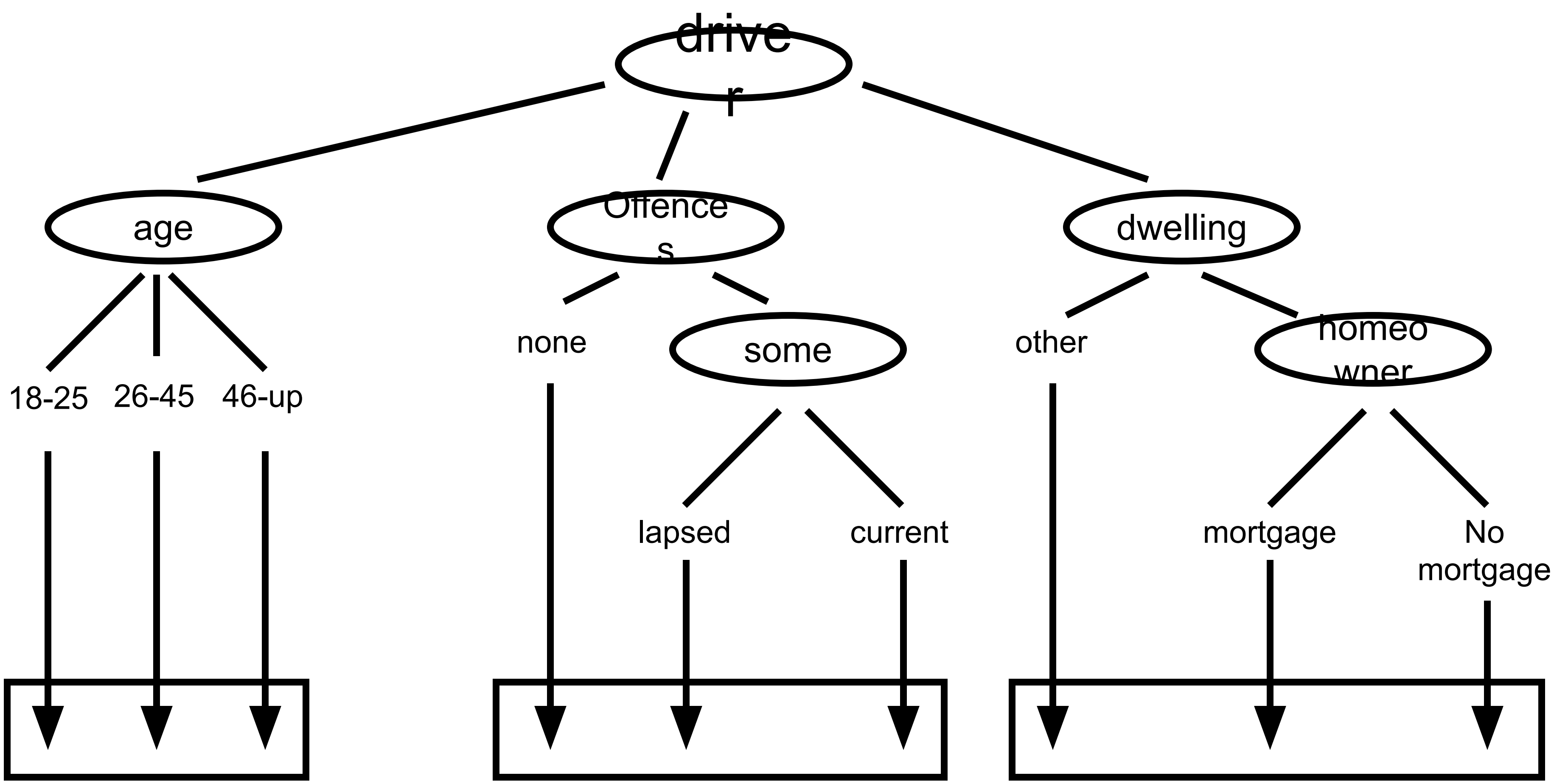
- Age
  - 18-25
  - 25-45
  - 46 upwards

- Offences
  - No past offences
  - Past offences
    - Lapsed offences
    - Current offences

- Dwelling status
  - Homeowner
    - Current mortgage
    - No mortgage
  - Other

- Etc. .....

Classification Tree – An Example

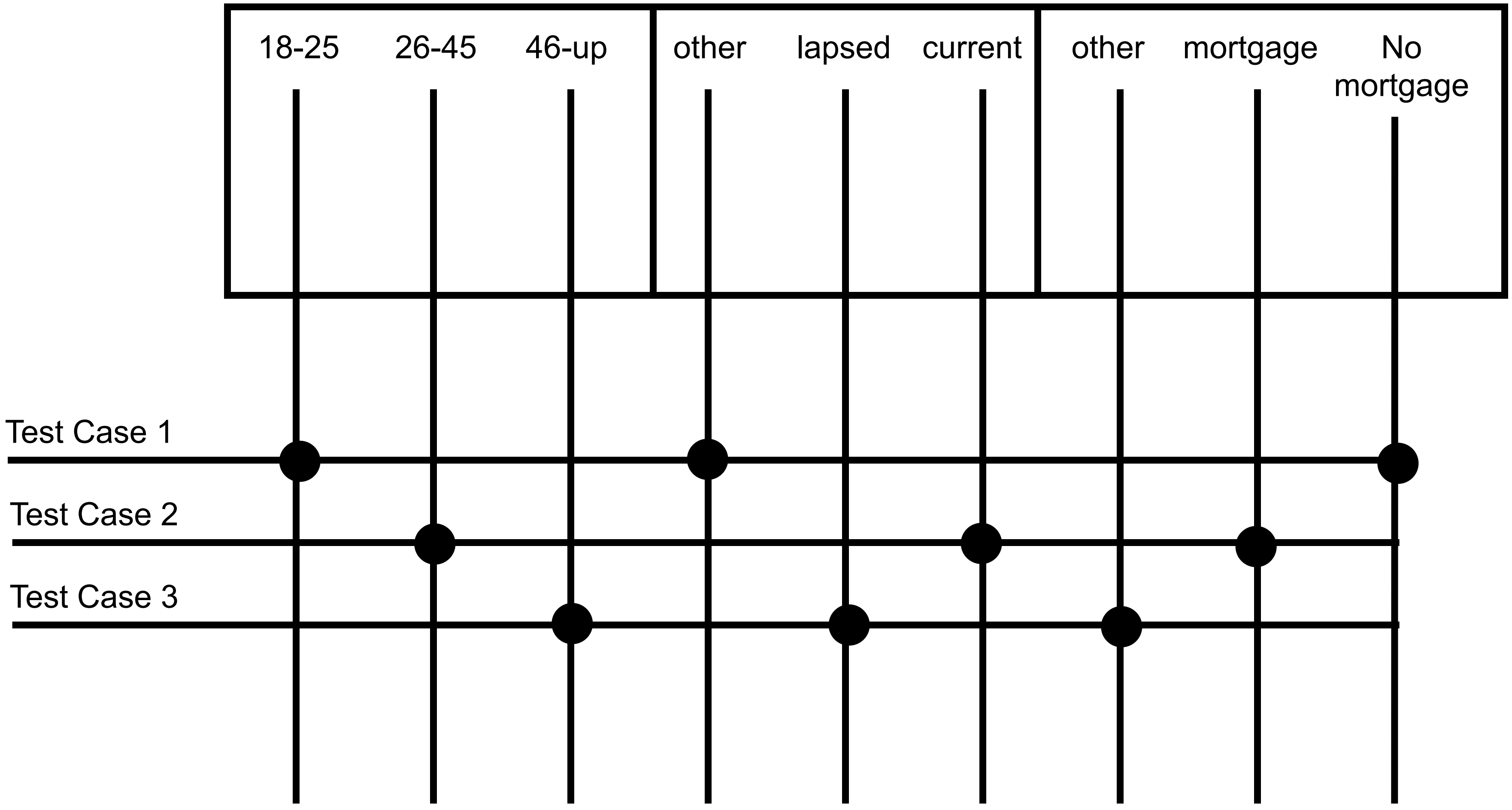


From classifications to test cases

- On the previous slide, there are five classes overall.
- Classes “Offences” and “Dwelling” have subclasses.
- The coverage target is to exercise all of the lowest level conditions.

- The classes identify which conditions are alternative
  - e.g. offences-none, some-lapsed and some-current are alternatives that cannot occur simultaneously.

From test conditions to test cases



Test Coverage

- The number of test case specifications and thus the scope of a test remain in principle for the user to decide.
- However, based on the Classification Tree, it’s possible for some values to be determined that provide clues to the number of test cases reasonably required.
- The first value is the number of test cases, if each leaf class is included at least once in a test case specification. This number is known as the <u>minimum criterion</u>.

Test Coverage

- The <u>maximum criterion</u> is the number of test cases that results when all permitted combinations of leaf classes are considered.

- A reasonable number of test case specifications obviously lies somewhere between the minimum and maximum criterion.

Test Coverage

- The objective of the Classification Tree Method is to determine a sufficient but minimum number of test case specifications.
- So generally speaking, it is not necessary to specify a test case for each possible combination. In fact, the Classification Tree Method should enable the user to use well-designed specifications, thus reducing the number of tests.
- The Classification Tree provides the necessary overview for this. In practical applications, this reduction of test cases is essential, since the maximum criterion can easily run into very high numbers.

What’s special about CTs?

- Simply a way of organising equivalence partitions and/or test values in a combinational way.
- The test case matrix is a more accessible way of identifying uncovered conditions, and duplicated test cases.
- Not really a method at all.
- You could do the same with a spreadsheet.

White Box Testing

Definition of White Box Testing

- Testing based on an analysis of the internal structure of the component or system.
- White Box Test Techniques include:
  - Statement Testing
  - Branch Testing
- Each of these techniques have an associated coverage measurement.

Remember – Coverage!

- **Coverage item:** An entity or property used as a basis for test coverage, e.g. equivalence partitions or code statements.
- **Coverage:** The degree, expressed as a percentage, to which a specified coverage item has been exercised by a test suite (i.e. a collection of test cases).
- **Coverage (measurement) Tool:** A tool that provides objective measures of what structural elements (e.g. statements or branches) have been exercised by a test suite.

Statement Testing and Coverage

- A white box test design technique in which test cases are designed to execute (executable) statements.

- Examples of executable statements:
  - assignments;
  - loops and selections;
  - procedure and function calls;
  - variable declarations with explicit initialisations;

- Examples of technique will be given in class.

Statement Coverage

- Every executable statement must be executed at least once (using the minimum number of test cases).

$$\text{Statement Coverage} = \frac{\text{number of statements executed}}{\text{total number of executable statements}} * 100$$

Statement Coverage

- Executable statements “do something”.
- ELSEs, ENDIFs etc. do not count as executable statements.

Branch Testing and Coverage

- **Branch Testing:** A white box test design technique in which test cases are designed to execute branches.

- **Branch Coverage:** The percentage of branches that have been exercised by a test suite. 100% branch coverage implies both 100% decision coverage and 100% statement coverage.

- Examples of technique will be given in class.

Branch Coverage

- Every branch is executed at least once for its True outcome, at least once for its false outcome, and very executable statement must be executed at least once (using the minimum number of test cases).

Branch Coverage

$$\text{Branch Coverage} = \frac{\text{number of executed branch outcomes}}{\text{total number of branch outcomes}} *100\%$$

McCabe’s Cyclomatic Complexity Metric

- The number of independent paths through a program. Cyclomatic complexity (V) is defined as: L – N + 2P, where

  - L = the number of edges/links in a graph

  - N = the number of nodes in a graph

  - P = the number of disconnected parts of the graph (e.g. a called method or function))

- Examples of approach will be given in class.

McCabe’s Cyclomatic Complexity Metric

Test Case Design and Coverage Techniques

References

- Standard for Software Component Testing
  - ©British Computer Society, SIGIST, 2001

    Working Draft 3.4 (27-Apr-01)

Techniques and Measures (Coverage)

- The Standard defines test case design techniques and test measurement (coverage) techniques.

- The techniques are defined to help users to
  - design test cases
  - and to quantify the testing performed (coverage).

- The definition of test case design techniques and measures provides for common understanding in both the specification and comparison of software testing.

Test Case Design and Measurement (Coverage) Techniques

- Black Box
  - Equivalence Partitioning
  - Boundary Value Analysis
  - Classification Trees

- White Box
  - Statement Coverage
  - Branch Coverage

Test Case Design and Measurement (Coverage) Techniques

- Each technique has its own “identified” coverage items.
- Coverage measurements determine which coverage items have been tested and which have not been tested.
- Usual to try and achieve 100% coverage for each technique.

Equivalence Partitioning – Design

- Equivalence partitioning is based on the premise that the inputs and outputs of a component can be partitioned into classes that, according to the component's specification, will be treated similarly by the component.
- Thus the result of testing a single value from an equivalence partition is considered representative of the complete partition.
- Both valid and invalid values are partitioned in this way.

Equivalence Partitioning

Black Box Technique

Equivalence Partitioning – Design

- Test cases shall be designed to exercise partitions.
- A test case may exercise any number of partitions. A test case shall comprise the following:
  - the input(s) to the component;
  - the partitions exercised;
  - the expected outcome of the test case.
- Test cases are designed to exercise partitions of valid values, and invalid input values.
- Test cases may also be designed to test that invalid output values cannot be induced.

Equivalence Partitioning – Coverage

- Coverage items are the partitions identified by the model.

- Coverage is calculated as follows:

$$\text{Equivalence partition coverage} = \frac{\text{(number of covered partitions)}}{\text{(total number of partitions)}} * 100\%$$

Equivalence Partitioning – Example

- Read the requirements.
- Divide the range of possible inputs (and outputs) into partitions / classes.
- For all values within a class, the software is expected to do “the same thing”.
- We test one value (at least) from each class.

Equivalence Partitioning – Example

- The valid range for the month is 1 to 12, representing January to December.
- This valid range is called a partition.
- In this example there are two further partitions of invalid ranges. The first invalid partition would be <= 0 and the second invalid partition would be >= 13.

Equivalence Partitioning – Example

```
  ... -2 -1 0   1 ............. 12   13 14 15 .....
--------------|-------------------|---------------------
invalid partition 1     valid partition          invalid partition 2
```

Equivalence Partitioning – Example

- Select one value from each partition to test the behaviour of the program. To use more or even all values of a partition will not find new faults in the program.

- The values within one partition are considered to be "equivalent". Thus the number of test cases can be reduced considerably.

- An additional effect of applying this technique is that you also find the so called "dirty" test cases. An inexperienced tester may be tempted to use as test cases the input data 1 to 12 for the month and forget to select some out of the invalid partitions. This would lead to a huge number of unnecessary test cases on the one hand, and a lack of test cases for the dirty ranges on the other hand.

Identifying equivalence partitions

Validity Rule	*Input Requirement*	*Valid Equivalence Class(es)*	*Invalid Equivalence Classes*
Range	Between 0.0 and 99.9	Between 0.0 and 99.9	<0.0 >99.9
Count	Must be 1, 11, 43, 82, 955	1, 11, 43, 82, 955	any number not in the valid list e.g. 2, 37, -56
Set	Must be BUS, CAR or TAXI	BUS CAR TAXI	e.g. XXX, TRAIN, PLANE
Must	Must begin with a letter	Begins with a letter (a-z, A-Z)	Any non alpha character or null e.g. 6, +, >, ~

Output partitions

- Output equivalence partitions are also important:
  - calculate tax brackets - can use different input salary ranges

Equivalence Partition Coverage

$$\text{Equivalence partition coverage} = \frac{\text{number of covered partitions}}{\text{total number of partitions}} * 100\%$$

Boundary Value Analysis

Black Box Technique

Boundary value analysis

- BVA is technique in which tests are designed to include representatives of boundary values.
- Experience shows more faults at boundaries of equivalence partitions where partitions are continuous.
- The maximum and minimum values of a partition are its boundary values.
- Don't forget test cases for output partitions

Boundary value analysis

- The expected input and output values should be extracted from the component specification.

- The input and output values to the software component are then grouped into sets with identifiable boundaries (using Equivalence Partitioning).
- Each set, or partition, contains values that are expected to be processed by the component in the same way.

Boundary Value Coverage

$$\text{Boundary value coverage} = \frac{\text{number of distinct boundary values executed}}{\text{total number of boundary values}} * 100\%$$

Basic Test Template

Test Template

- **Test Identifier** - Unique number to identify test.
- **Test Objective** - Objective of test.
- **Test Input(s)** - Input values to drive the test.
- **Expected Output(s)** - Expected Output(s) to be generated if the test is a success.

Classification Trees

Boundary Value Analysis – Example

```
~ ... -2 -1 0   1 .............. 12   13 14 15 ..... ~
---------------|--------------------|----------------------
invalid partition 1     valid partition        invalid partition 2
```

- What about the boundaries at ~ ?

What is the Classification Tree Method (CTM)?

- Testing is a step in the software development process. However, the planning of such testing often raises the same questions:
  - How many tests should be run?
  - What test data should be used?
  - How can error sensitive tests be created?
  - How can redundant tests be avoided?
  - Have any test cases been overlooked?
  - When is it safe to end testing?
- The Classification Tree Method offers a systematic procedure to create test case specifications based on a problem definition.

Using Classification Trees

- Black box test “technique” aimed at preparing test cases.
- Appropriate where functionality depends on combinations of conditions.

- CTs useful in any ‘combinatorial’ situation.

An Example

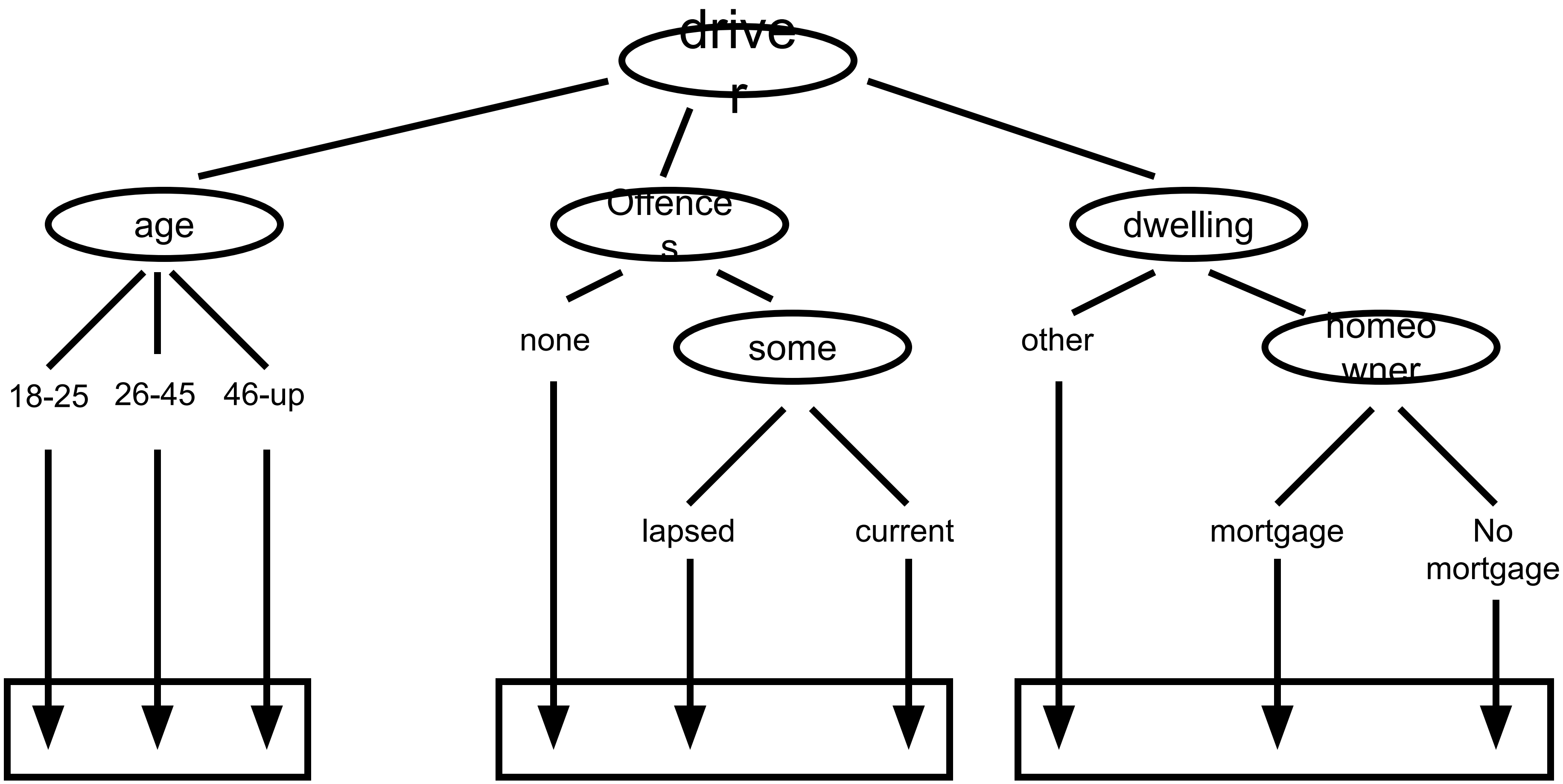
- Age
  - 18-25
  - 25-45
  - 46 upwards

- Offences
  - No past offences
  - Past offences
    - Lapsed offences
    - Current offences

- Dwelling status
  - Homeowner
    - Current mortgage
    - No mortgage
  - Other

- Etc. .....

Classification Tree – An Example

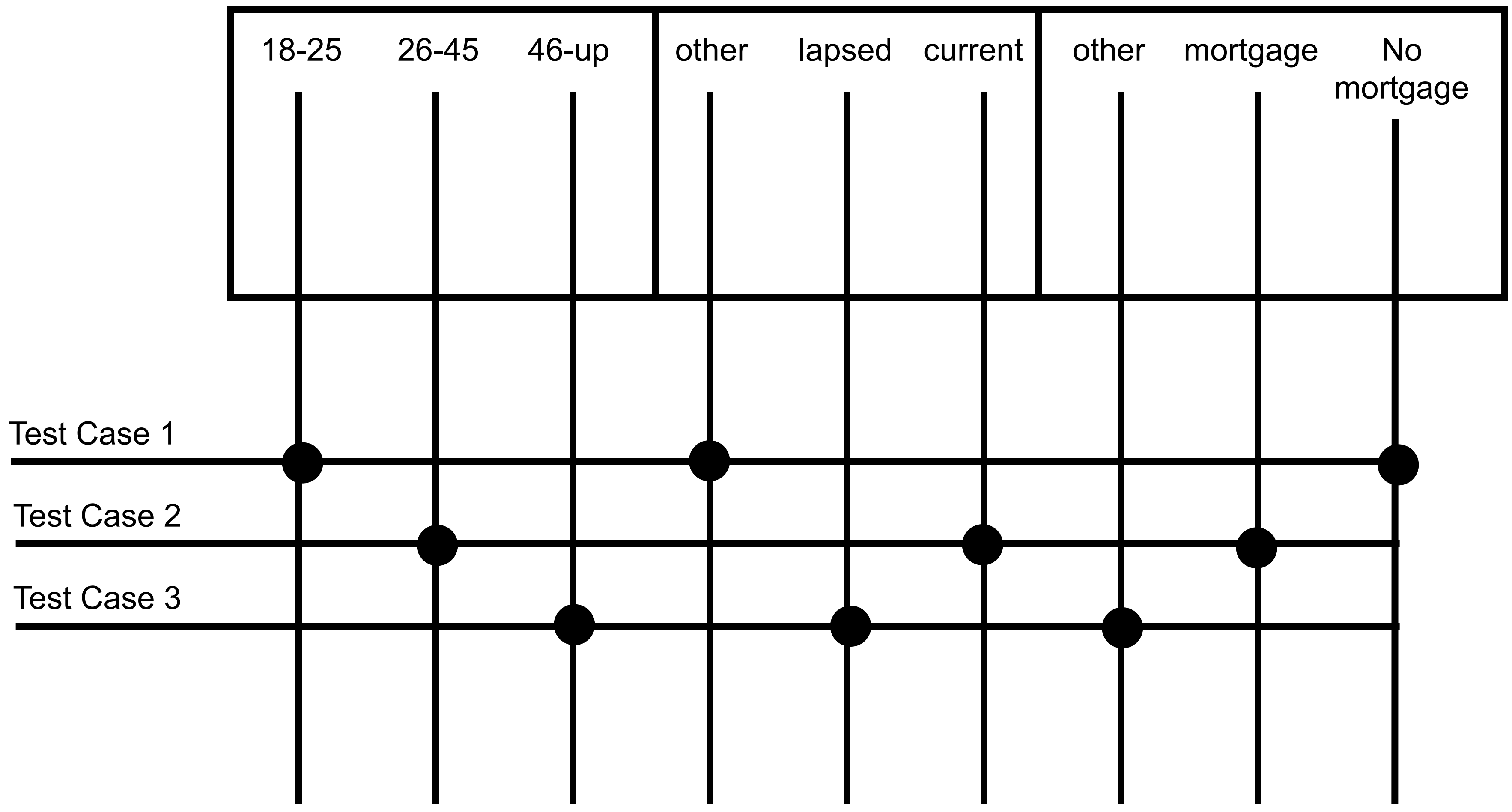


From classifications to test cases

- On the previous slide, there are five classes overall.
- Classes “Offences” and “Dwelling” have subclasses.
- The coverage target is to exercise all of the lowest level conditions.

- The classes identify which conditions are alternative
  - e.g. offences-none, some-lapsed and some-current are alternatives that cannot occur simultaneously.

From test conditions to test cases



Test Coverage

- The number of test case specifications and thus the scope of a test remain in principle for the user to decide.
- However, based on the Classification Tree, it’s possible for some values to be determined that provide clues to the number of test cases reasonably required.
- The first value is the number of test cases, if each leaf class is included at least once in a test case specification. This number is known as the minimum criterion.

Test Coverage

- The maximum criterion is the number of test cases that results when all permitted combinations of leaf classes are considered.

- A reasonable number of test case specifications obviously lies somewhere between the minimum and maximum criterion.

Test Coverage

- The objective of the Classification Tree Method is to determine a sufficient but minimum number of test case specifications.
- So generally speaking, it is not necessary to specify a test case for each possible combination. In fact, the Classification Tree Method should enable the user to use well-designed specifications, thus reducing the number of tests.
- The Classification Tree provides the necessary overview for this. In practical applications, this reduction of test cases is essential, since the maximum criterion can easily run into very high numbers.

What’s special about CTs?

- Simply a way of organising equivalence partitions and/or test values in a combinational way.
- The test case matrix is a more accessible way of identifying uncovered conditions, and duplicated test cases.
- Not really a method at all.
- You could do the same with a spreadsheet.

Statement Coverage

White Box Technique

Statement Coverage

- Every executable statement must be executed at least once (using the minimum number of test cases).

$$\text{Statement Coverage} = \frac{\text{number of statements executed}}{\text{total number of executable statements}} * 100$$

Statement Coverage

- Executable statements “do something”.
- ELSEs, ENDIFs etc. do not count as executable statements.

Branch Coverage

White Box Technique

Branch Coverage

- Every branch is executed at least once for its True outcome, at least once for its false outcome, and very executable statement must be executed at least once (using the minimum number of test cases).

Branch Coverage

$$\text{Branch Coverage} = \frac{\text{number of executed branch outcomes}}{\text{total number of branch outcomes}} * 100\%$$

White Box Testing

Definition of White Box Testing

- Testing based on an analysis of the internal structure of the component or system.
- White Box Test Techniques include:
  - Statement Testing
  - Branch Testing
- Each of these techniques have an associated coverage measurement.

Remember – Coverage!

- **Coverage item:** An entity or property used as a basis for test coverage, e.g. equivalence partitions or code statements.
- **Coverage:** The degree, expressed as a percentage, to which a specified coverage item has been exercised by a test suite (i.e. a collection of test cases).
- **Coverage (measurement) Tool:** A tool that provides objective measures of what structural elements (e.g. statements or branches) have been exercised by a test suite.

Statement Testing and Coverage

- A white box test design technique in which test cases are designed to execute (executable) statements.

- Examples of executable statements:
  - assignments;
  - loops and selections;
  - procedure and function calls;
  - variable declarations with explicit initialisations;

- Examples of technique will be given in class.

Branch Testing and Coverage

- **Branch Testing:** A white box test design technique in which test cases are designed to execute branches.
- **Branch Coverage:** The percentage of branches that have been exercised by a test suite. 100% branch coverage implies both 100% decision coverage and 100% statement coverage.
- Examples of technique will be given in class.

McCabe’s Cyclomatic Complexity Metric

- The number of independent paths through a program. Cyclomatic complexity (V) is defined as: L – N + 2P, where

  - L = the number of edges/links in a graph

  - N = the number of nodes in a graph

  - P = the number of disconnected parts of the graph (e.g. a called method or function))

- Examples of approach will be given in class.

Meetings or Ceremonies

Meetings or Ceremonies

- Since the teams are working hard during the sprint or iteration to design, code, and test the requirements, we need a mechanism to keep track of their progress and make sure everything is working well on the teams.
- Similar to how the XP tracker asks each developer how things are going, Scrum, Lean, and DSDM all recommend a daily stand-up meeting, sometimes called the daily Scrum.

Daily Stand-Up Meeting

- Development teams use the **daily stand-up meeting** to do a quick status check and prepare the team for the day.

- These meetings usually take less than 15 minutes and provide an opportunity to understand the team’s progress in the iteration.

- Each team member answers three key questions during the daily stand-up meeting:

Daily Stand-Up Meeting

- **What did I do yesterday?** This is an opportunity for a developer to share what tasks were closed yesterday or if something was more difficult than expected or will require more time.
- **What am I planning to do today?** This information helps the team know the developer's area of focus for the day and also if he or she will be unavailable for some portion of time because of meetings or personal considerations. By understanding what a teammate is working on, the other team members can align their work, if necessary.
- **Is there anything blocking my progress?** This could be anything that is preventing advancement, from a technical issue to the lack of a test environment to a clarification question with the product owner. The key is to share challenges to keep things moving forward in the iteration. The output of this discussion serves as the action items for the Scrum master.

Daily Stand-Up Meeting

- Some people view the daily stand-up as a status meeting, but it is much more than that.
- A status meeting is simply informing others on your progress in a static and single-threaded way, but, the daily stand-up meeting is an opportunity for team calibration.
- If one team member reports struggles with a task, then other team members can jump in and help.

Daily Stand-Up Meeting

- This help might be sitting together to solve the problem or offering to take ownership of other tasks to allow the teammate to continue working on the troublesome task.
- Within Agile, the team succeeds or fail together. To ensure that commitments are kept, the team needs to band together to deliver on all tasks.
- The daily stand-up meeting provides a forum for teammates to calibrate on the work remaining and support each other to success.

Daily Stand-Up Meeting

- There are a few other important parameters for the daily stand-up meeting.
- First, the meeting is open to everyone, meaning that stakeholders and extended team members are welcome to attend, but only the “pigs” may talk.
- The reason for this distinction relates to the goal of the meeting—*it is about team calibration*.

Daily Stand-Up Meeting

- Although a stakeholder may have an opinion to share, the daily stand-up meeting is probably not the right forum.
- Also, because the daily stand-up is designed as a short meeting, where the participants actually stand up to prevent the meeting from lasting longer than necessary, it is not the right place for problem solving.
- When a developer or tester is answering the three questions, it is often easy to start brainstorming and problem solving *as part of the discussion*.
- The Scrum master needs to control the meeting and push certain discussions “offline,” meaning that they should be discussed after the meeting.

Daily Stand-Up Meeting

- When introducing the daily stand-up to new Agile teams, there can be hesitation about a daily meeting.

- People often believe they do not have time to meet every single day, and they initially view it as unnecessary overhead.

- However, if the daily stand-up meeting is run well, with a focus on information sharing and team calibration, it can accelerate a team toward meeting their goals.

- High-performing teams often find that other longer and more tedious meetings can be avoided by conducting an effective daily stand-up.

Sprint Review or Demo

- A Sprint review, or Sprint demo, is a meeting hosted by the team that is open to a wide audience including extended team members, stakeholders, and potentially even external customers.

- The sprint review provides the opportunity to showcase the working software to inform and educate the organization on what has been completed and to gather feedback to ensure that the product is delivering on expectations.

Sprint Review or Demo

- The session is not intended to be a large, professional meeting with a lengthy presentation.

- In fact, most teams limit the “prep” time for the demo to two hours or less; we want the teams to spend their time writing great code, not producing pretty PowerPoint slides.

Sprint Review or Demo

- The person who leads the discussion varies by team and may even vary by Sprint review.
- Often the product owner will describe the user stories that were tackled during the sprint, and the Scrum master will demonstrate the working software.
- On other teams, the product owner runs the entire meeting, because having a nontechnical person work through the software demonstrates its usability.
- On still other teams, the developers do the demonstration because they feel a great sense of ownership for the value that they have created with the software.

Sprint Review or Demo

- Within Extreme Programming (XP), many teams have the customer (product owner) lead the demonstration because the entire team wants to showcase their work, and the customer should demonstrate the progress that has been made on the features.

Sprint Review or Demo

- If the feedback is particularly negative, the team will need to evaluate whether they should make the new feature publicly available.

- Positive feedback validates that the new feature is ready for deployment.

Sprint Review or Demo

- One of the product owner’s responsibilities is to “accept or reject” the completed user stories; this should be done either before or at the Sprint review meeting.

- If the stakeholders offer opportunities for improvement, those can be documented as potential new requirements.

- The product owner is responsible for collecting all of the feedback and determining what items will be addressed in future sprints and their relative priority

Sprint Review or Demo

- Stakeholders should not assume that all feedback will be incorporated.

- The product owner should evaluate the business value of each suggestion and compare it against the items in the product backlog.

Retrospectives

- Retrospectives are an important part of Agile software development as a forum for open communication among team members and as a tool for continual improvement.
- The Agile Manifesto emphasizes team retrospectives in this principal: “At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.”

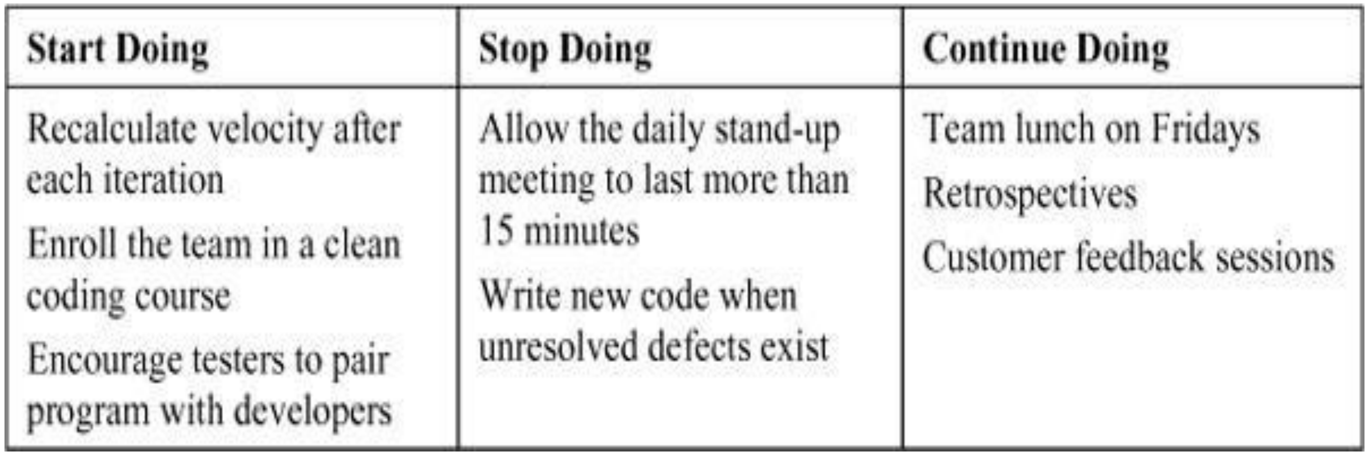
Retrospectives

- A **retrospective** is a team meeting at the end of an iteration where the team has the opportunity to inspect and adapt their teamwork.

- Agile’s concept of team self-organization, collaboration, and continuous improvement makes the retrospective a critical meeting to have for the growth and development of the team.

- One method for facilitating the discussion is to pass out note cards and ask participants to jot down their observations into categories; the categories could be “stop doing,” “start doing,” and “continue,” as in the following table.

Retrospectives



Start Doing	Stop Doing	Continue Doing
Recalculate velocity after each iteration Enroll the team in a clean coding course Encourage testers to pair program with developers	Allow the daily stand-up meeting to last more than 15 minutes Write new code when unresolved defects exist	Team lunch on Fridays Retrospectives Customer feedback sessions

http://www.mountaingoatsoftware.com/agile/scrum/sprint-retrospective

Retrospectives

- Retrospectives allow the teams to discuss the workflow and how that can be enhanced, but the bigger value comes in the sense of teamwork and trust that is emphasized during this meeting.
- Gathering the team together to talk about how they are getting along can be awkward and uncomfortable, but it is the surest way to improve things.

- There are a number of ways to manage retrospectives to make the conversation more natural, while ensuring that the key concerns are addressed effectively.

Retrospectives

- One method is called “Prioritizing with Dots” and it helps to prioritize the items that the team wants to work on. The steps involved are:

- Have team members write down specific action items that they would like the team to address on a post-it note, note card, white board, etc.

- Post each suggestion with no judgment or priority. After every idea is on the board, give each participant three small stickers; these could be small post-its, colored dots, or even labels from the printer.

- Each person places the three stickers on any action item(s) he or she considers most important; all three stickers could be put on a single item, for example, or they could put one each on three different items.

Retrospectives

- When the exercise is completed, the team knows that the action item with the most stickers is the top priority.
- Based on the sticker allocation, you can decide how many action items you want to address.
- There is nothing magical about the number three—each person could vote five times or eight; teams should use whatever number works best to achieve their goals.

Retrospectives

- Retrospectives are an important vehicle for the team to continuously grow together as an entity, which will make their work environment more productive and more enjoyable.

Measuring Success in Agile

- How do you know if an Agile project is successful?

- This topic has sparked great debate, because some Waterfall metrics drive the wrong behavior and thus are inappropriate.

- The following framework for measuring performance, suggests four categories of measurement:
  1. Do it fast
  2. Do it right
  3. Do it on time
  4. Keep doing it

Measuring Success in Agile

- These parameters align well with the core values and principles of Agile.
- In the “do it fast” category, we can measure the teams’ productivity and their responsiveness.
- In the “do it right” category, we are measuring quality and customer satisfaction.
- When we “do it on time,” we demonstrate a predictable cadence for when working software is delivered.
- Under “keep doing it,” we want to measure employee satisfaction, which is a critical component of sustainability as referenced in the Agile principles.

Measuring Success in Agile

- Truly, the absolute best measure of success for Agile teams is customer satisfaction.
- If your organization is delivering high-quality software that meets the needs of the users and is delivered quickly and predictably, then your customer satisfaction scores should be steadily (or dramatically) improving.

Conclusion

Conclusion

- Tracking and reporting are critical to Agile projects because of the focus on transparency and continuous improvement.

- Tracking mechanisms (see next point) within all of the agile methodologies will help the teams determine when and how to adjust to ensure delivery on their commitments.

- Burn charts, information radiators, FDD parking lots, and quality tracking measures all provide valuable information to the teams – To be discussed later!

Conclusion

- The meeting structure also reinforces the tracking and reporting measures, with the daily stand-up providing real-time feedback on progress and allowing the team to calibrate on the remaining work.

- The Sprint demo or review is a forum for stakeholders to see the working software and provide feedback.

Conclusion

- Finally, the Sprint retrospective helps the team to inspect and adapt their relationships and team effectiveness.
- It is important to measure the overall performance of an Agile project, and customer satisfaction is a meaningful way to ensure that the organization is delivering business value to the marketplace.

Kanban Method

Kanban

- Focus – Just-in-time development and evolving the process to create an optimal system.

- The word *Kanban* loosely translates to “signboard” in Japanese.

- A **signboard** is traditionally described as a visual process-management system that tells teams what to produce, when to produce it, and how much to produce.

Kanban

- The visual signboard and work-in-progress pull method, also known as the **Kanban system**, is the most popular tool that organizations adopt when they use the Kanban method.

Kanban – Key Principles

- The Kanban method is based on four key principals.

1. **Foster leadership at all levels of the organization—**Everyone in the organization is encouraged to act as leaders, from the entry-level employee to the executive team.

2. **Start with what you know—**Kanban is not prescriptive in nature and assumes that organizations are not all the same. Instead, it is important to understand where you are today and use change management approaches to evolve from that point.

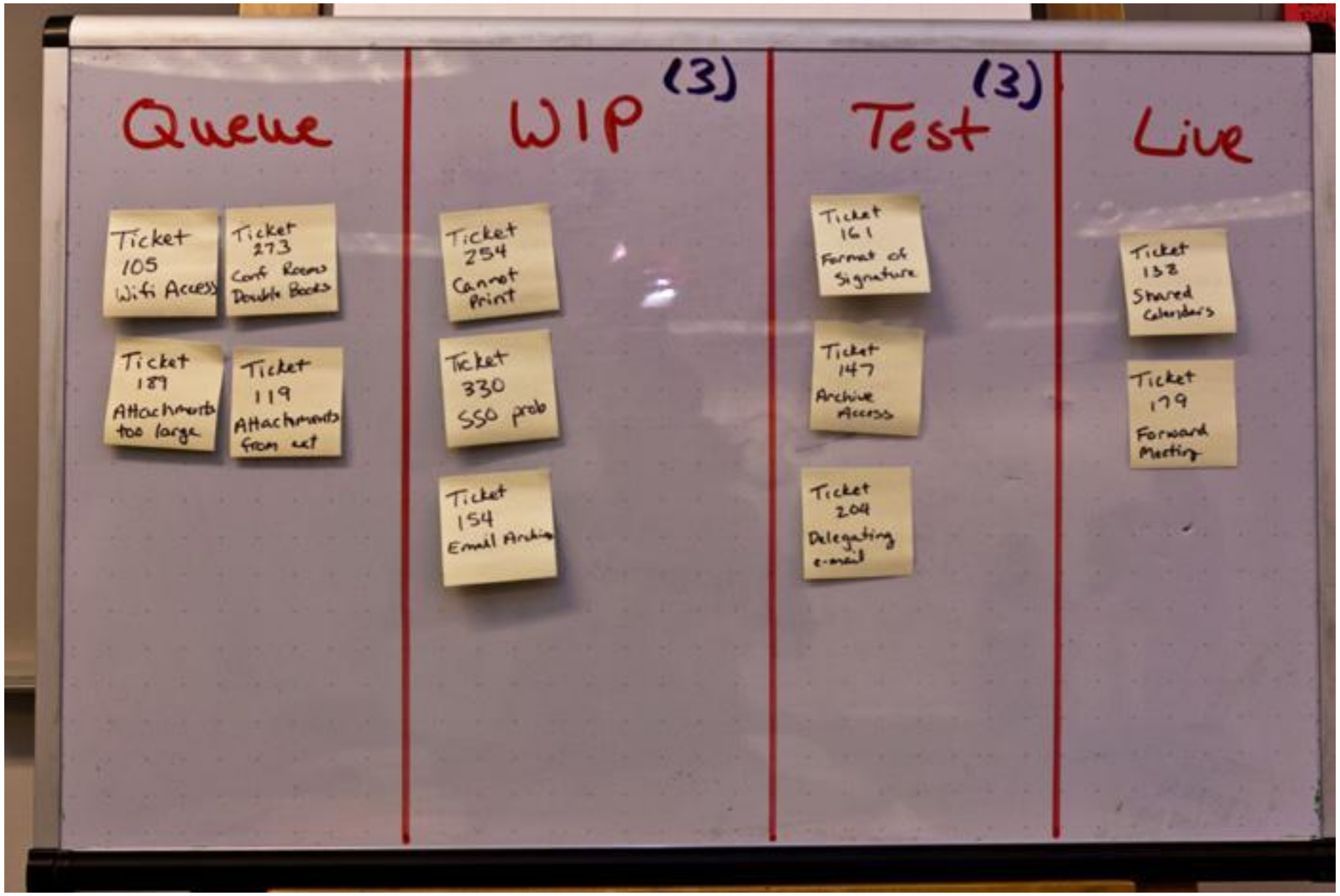
Kanban – Key Principles

3. **Focus on incremental and evolutionary change—**The goal is not to overhaul the process, culture, and product overnight. Instead, it is important to make small but impactful changes often.
4. **Respect current methodologies and roles—**The current process was put there for a reason; something about it has worked, and that is why people have continued to use it. It is important to preserve what has worked and to change what is no longer helping the team achieve their goals.

Kanban – Core Practices

1. **Visualize workflow—**Teams need to visually understand their work so that they can optimize their workflow. The Kanban method often uses a board with columns and cards to show the workflow (see picture on next slide).
2. **Limit work in progress, and pull in new work only when time becomes available—**The team cannot be efficient and focused if too much work is on their plate; it is important to focus on one task at a time and not to start a new task until the current task is completed.

Kanban Board


Queue
WIP (3)
Test (3)
Live
Ticket 105 Wifi Access
Ticket 273 Conf Rooms Double Books
Ticket 189 Attachments too large
Ticket 119 Attachments from ext
Ticket 254 Cannot Print
Ticket 330 SSO prob
Ticket 154 Email Archiv
Ticket 161 Format of Signature
Ticket 147 Archive Access
Ticket 204 Delegating e-mail
Ticket 138 Shared Calendars
Ticket 179 Forward Meeting

Kanban – Core Practices

3. **Manage the flow through the system—**Just as with a relay race, the team needs to understand how the work progresses toward the goal. The team needs to ensure that the appropriate level of work is flowing through the system and that bottlenecks are being kept to a minimum.
4. **Provide explicit policies—**The team needs to understand how to work within their organization in order to eliminate ambiguity. For example, a team may enact the policy that they do not pull a new feature until the current feature has been through unit test.

Kanban – Core Practices

5. **Improve collaboratively and evolve experimentally—**A team can best improve if members first have a shared vision and work together to generate ideas for improvement. Evolution cannot happen through guesswork or intuition; the team needs to try new approaches and objectively evaluate their merit.
6. **Ensure feedback is part of the process—**Both positive and negative feedback are essential in understanding how the organization needs to change and evolve.

Kanban - Summary

- One important difference of Kanban from other key approaches such as XP or Scrum is that it does not rely on iterations.
- Kanban allows the team to focus by limiting the work that is in progress and advocates a continuous flow of work.
- Kanban does not prohibit the use of iterations or time boxes, and it is common for teams to use these tools with Kanban, but specific time frames are not considered a necessary element of this approach.

Tracking and Reporting

Introduction

- All of our efforts to prepare the requirements plan and groom the activities and test our code are critical steps toward producing quality software that will meet the customer’s needs and deliver business value.

- This section is focused on tracking and measuring our success and quickly identifying the areas that need attention and improvement.

Introduction

- We review the tools that are used in tracking, such as Kanban task boards, burn charts, information radiators, and progress charts.
- We show how quality is built into the process and then discuss the meetings or ceremonies where progress is measured and adjustments are made, if necessary.
- We explain how to define success within Agile, including the ever-important metric of customer satisfaction.

Kanban

- Kanban is a methodology that we have not focused on because it is fundamentally different from the others that we have discussed.
- Most of the Agile methodologies deliver working software in a time-boxed fashion, meaning that the work proceeds in iterations, or sprints.
- Conversely, Kanban operates in a “continuous flow” model, meaning that there is no time box for development; tasks are continuously added to the backlog and removed as they are completed

Kanban

- Kanban has three primary characteristics:
  - **Visualize the workflow**
    - A Kanban board maps the steps of the workflow in columns across a board.
    - Items, or tasks, are represented on cards or sticky notes and are tracked as they move through the workflow steps.

Kanban

- **Limit work in progress (WIP)**
  - Assign limits to each column, or workflow step, so that no one group or person can be overloaded with work.
  - The WIP limits naturally identify bottlenecks in the process so they can be addressed.
- **Measure the lead time**
  - By understanding the workflow and removing the bottlenecks, the teams will discover the time it takes for a task to move from creation to resolution, thus providing the organization with metrics for lead time.

Kanban vs Scrum

- How does Kanban compare with Scrum?
- Scrum is a great fit for products/projects with a time-boxed workflow, such as product development efforts, that can progress in a series of sprints or iterations.
- On the other hand, Kanban is a better fit for an unpredictable workflow, unplanned work, and development tasks that require deployment in less time than a typical iteration.
- For example, Kanban is a great fit for helpdesk tickets and custom software support where small tickets are continually created and require speedy resolution.

Kanban vs Scrum

- This is where Kanban is a great option because it shares all of the Agile elements that make this work so well—collaboration, clear prioritization, small increments of work, fast delivery of working software—and moves it to a continuous workflow.

- It helps to separate new product development from support of the current product.

Kanban Board

- Teams often use white boards and sticky notes to create the Kanban board, but any visual tool can be used, from a chalkboard to a large sheet of paper and markers.
- Looking at a specific example, we have five columns on our Kanban board: “backlog,” “not started,” “in progress,” “testing,” and “completed.”
- Since workflow stages can vary based on the nature of the work, the columns will change to match the needs of the particular team.

Kanban Board – An Example

- **Backlog**—The product owner puts all of the requirements (stories) on sticky notes in priority order. The estimated effort of the requirement is also noted. This list could be quite lengthy, depending on the size of the backlog.

- **Not started**—These stories have been selected by a developer and are therefore assigned, but the development work has yet to begin. This is equivalent to a “pending” queue, or the next items to be worked.

- **In progress**—These are the requirements that are currently being coded by a developer.

- **Testing**—This is when the testing is done to ensure that the code is working as intended.

- **Completed**—The requirements in this column are considered ready to be delivered to a customer.

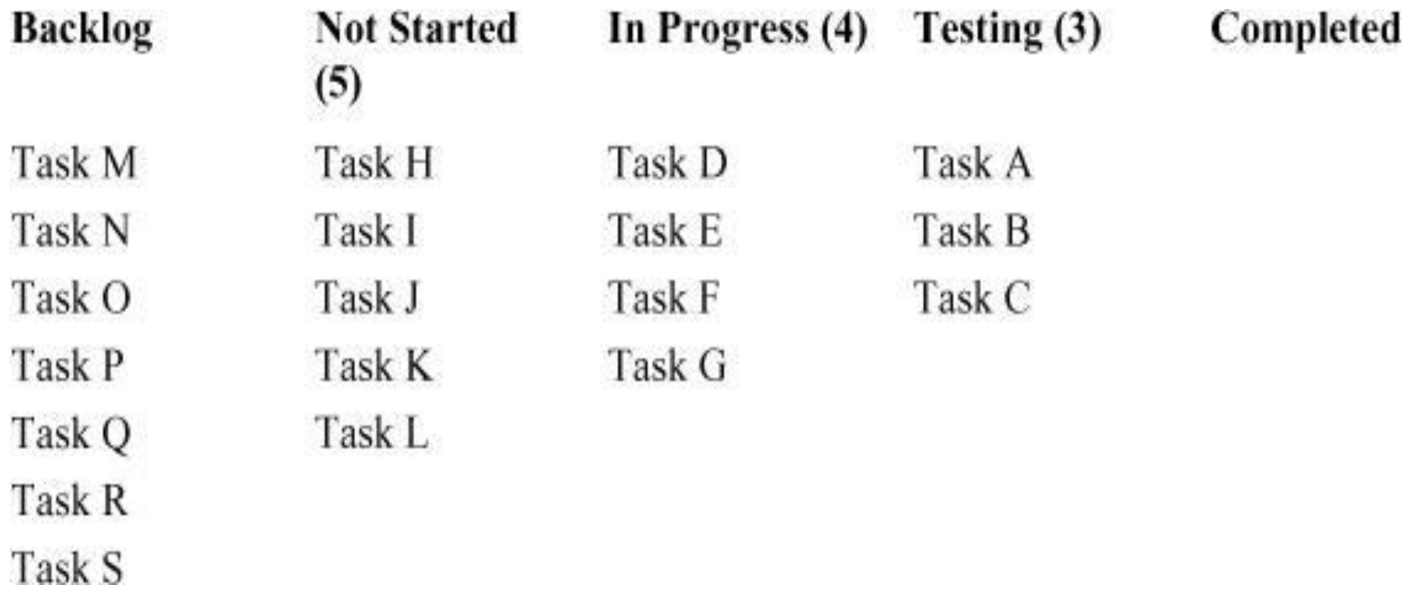
Kanban – Work In Progress (WIP) Limit

- Kanban allows for limiting the amount of work (stories) in a given stage at any point in time; this limit is commonly referred to as the work in progress (WIP) limit.
- WIP limits help organizations to identify where there are bottlenecks in their processes.
- One of the key tenets of Agile is increased transparency, and the Kanban WIP limits bring attention and focus to the areas of the business where things are slowing down so we can quickly address and correct the situation.

Kanban – Work In Progress (WIP) Limit: An Example

- For example, in a Kanban board, we have the following WIP limits:
  - Backlog: No limit
  - Not started: WIP limit of 5
  - In progress: WIP limit of 4
  - Testing: WIP limit of 3
  - Completed: No limit
- By having these limits, we can identify where our bottlenecks are.

Kanban – Work In Progress (WIP) Limit: An Example



Backlog	Not Started (5)	In Progress (4)	Testing (3)	Completed
Task M	Task H	Task D	Task A	
Task N	Task I	Task E	Task B	
Task O	Task J	Task F	Task C	
Task P	Task K	Task G		
Task Q	Task L			
Task R				
Task S				

In this example, all of the WIP limits have been reached, and each person is working diligently on his or her tasks.

Kanban – Work In Progress (WIP) Limit: An Example

- If the developer finishes task D, he cannot move it into testing because their queue is already full.
- He must wait until task A, B, or C is completed in testing and moved to completion before task D can be pulled into the testing queue.
- To move task D into testing now would violate the testing WIP limit of 3.
- Therefore, our developer who has completed task D cannot pull task H in from the “not started” queue, because adding that task to the “in progress” queue would violate the WIP limit of 4.

Kanban – Work In Progress (WIP) Limit: An Example

- There are many reasons why this is valuable and reinforces the values of Agile.
- Perhaps the developers are moving very fast and, as an unfortunate result, they are writing buggy code. Thus, the testing queue is backed up because they cannot get tasks A, B and C to pass testing.
- The developers are finishing more tasks but they cannot move them to testing yet—and they cannot pick up any new tasks.
- Therefore, the developers can help the testers with their tasks; the testing queue will get cleared, and the developers will understand that the better code they write, the more smoothly the process will go.

Kanban – Work In Progress (WIP) Limit: An Example

- The WIP limits also prevent the product owner (or executives) from forcing too much work on the team.
- The “not started” queue has a WIP limit of 5, so those represent the stories or tasks that are well defined and prioritized.
- Just as Scrum limits the amount of work by allowing the team to commit only to what they can reasonably finish in the duration of a sprint, the WIP limits accomplish the same goal by limiting how much work can be in each queue.

Kanban – Work In Progress (WIP) Limit

- In this example, WIP limits are set by the number of tasks, but as you can imagine, not all tasks are equal.
- Some tasks are much larger or more complex than others, so some teams set their WIP limits by the number of story points.
- Setting and enforcing WIP limits can be challenging for teams, so some teams that are new to Kanban start with the workflow and the Kanban board and add WIP limits later, when they are more comfortable with the process.

Kanban – Work In Progress (WIP) Limit

- The WIP limits also help with the third characteristic of Kanban—the ability to measure lead time.
- Just as a Scrum team will establish their velocity, Kanban teams need to understand the time it takes for a task to flow from beginning to end.
- The team could then predict that if a task is accepted into the development queue on Tuesday, it will typically be released on Friday, barring any unforeseen circumstances.

Kanban

- Kanban is a wonderful alternative for teams who want to embrace all of the values and principles of Agile but whose work is unpredictable and difficult to plan.
- In many organizations, both Scrum and Kanban are utilized, depending on the makeup of the work.

Tracking

Tracking

- Regardless of the particular methodology being used, tracking progress is critical to Agile.

- Increased transparency and the desire to eliminate surprises make tracking essential to effectively managing an Agile project.

Extreme Programming (XP)

- XP takes tracking so seriously that one of their roles is that of a tracker.

- The tracker talks to the developers daily and asks two questions—how many ideal days have they worked on a task, and how many ideal days do they have remaining.

- Remember that ideal days or ideal time is the amount of time a task would take under ideal circumstances, with no interruptions, phone calls, meetings, etc.

Extreme Programming (XP)

- XP emphasizes that the tracker should talk to each developer about the task status.

- This reinforces the importance of face-to-face communication so the tracker can ensure that he or she obtaining accurate and realistic responses from the developer.

- Scrum addresses this in a similar fashion, through the use of the daily stand-up meeting (described later in more detail).

Burn Charts

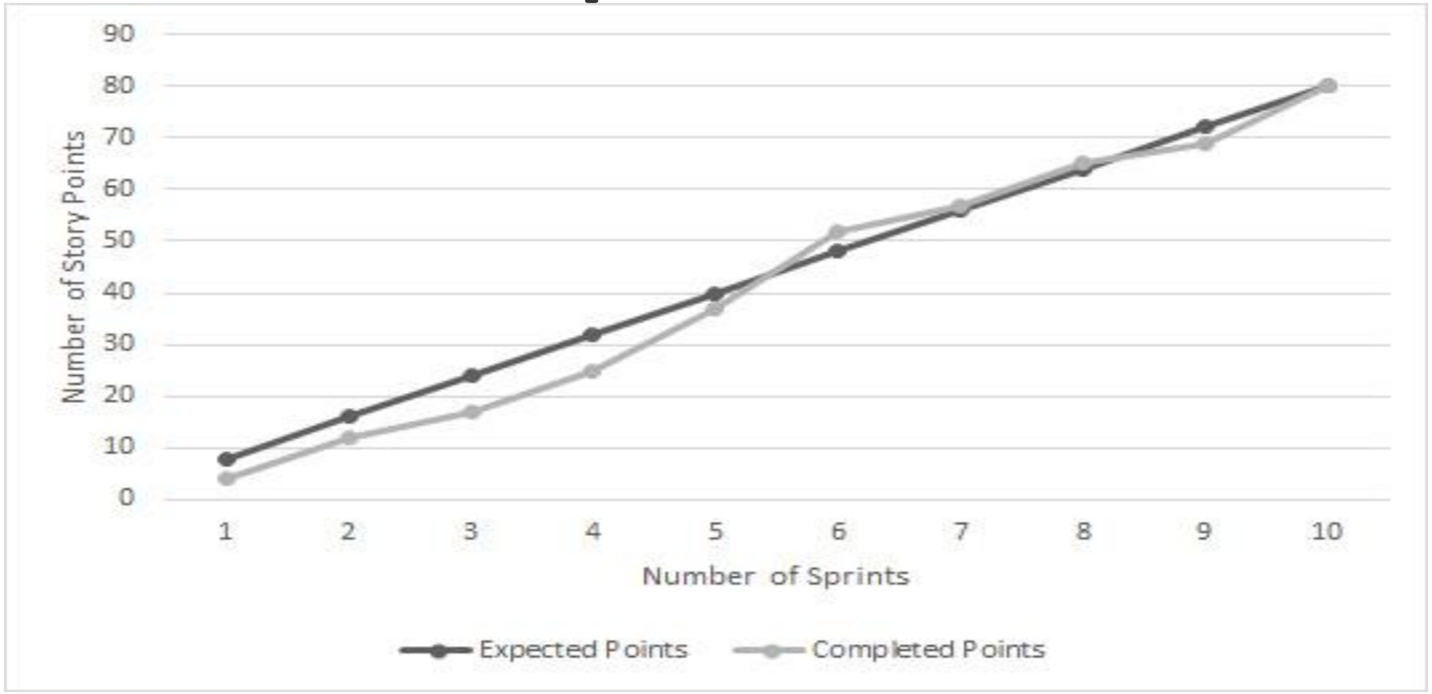
- Burn charts are a mechanism to be transparent about the progress of a release or iteration so the team can decide if they need to alter their approach to the work in order to honor their commitments.

- By being able to easily see their actual progress charted against the planned progress, they have actionable data to inform their decisions

Burn-up Charts

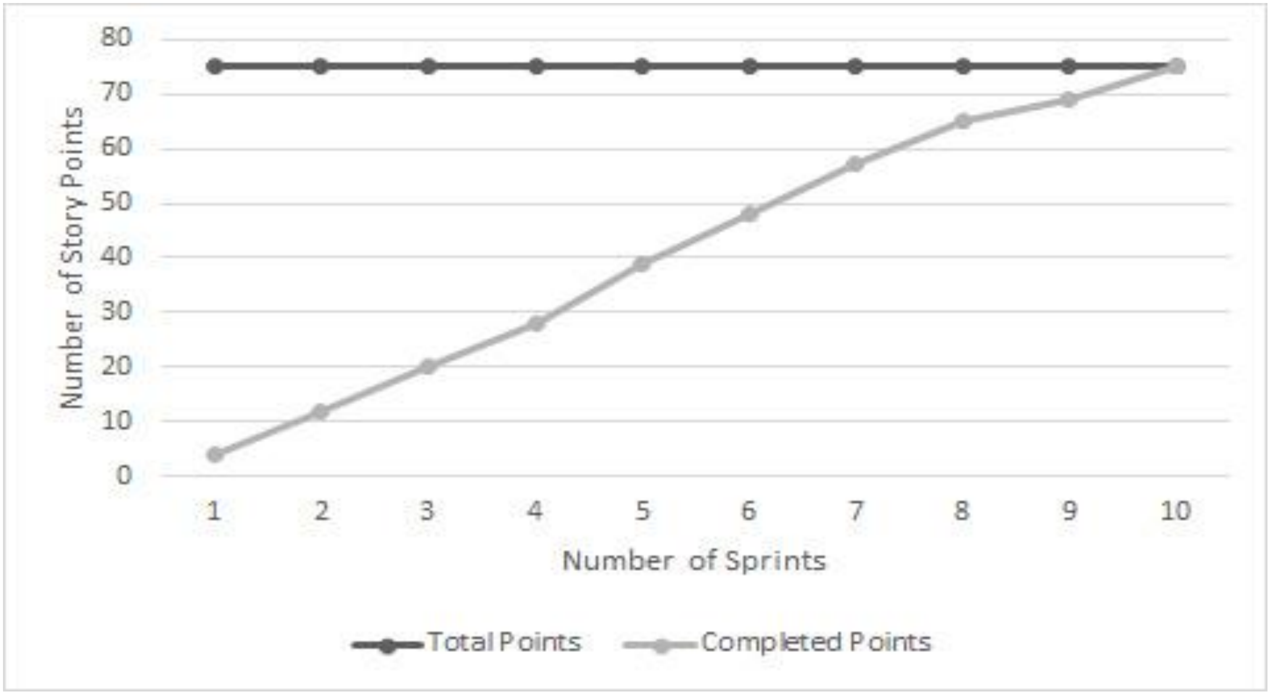
- Burn-up charts are a way to depict progress toward a product’s release goal.
- Typically, time is on the X axis, either in sprints, months, or quarters, and the Y axis represents the release at completion.
- The Y axis can be either story points or feature descriptions.

Example burn-up chart by expected points



- The black line is the cadence that we expect to see relative to the number of story points delivered in each sprint.
- The grey line represents actual story points delivered.

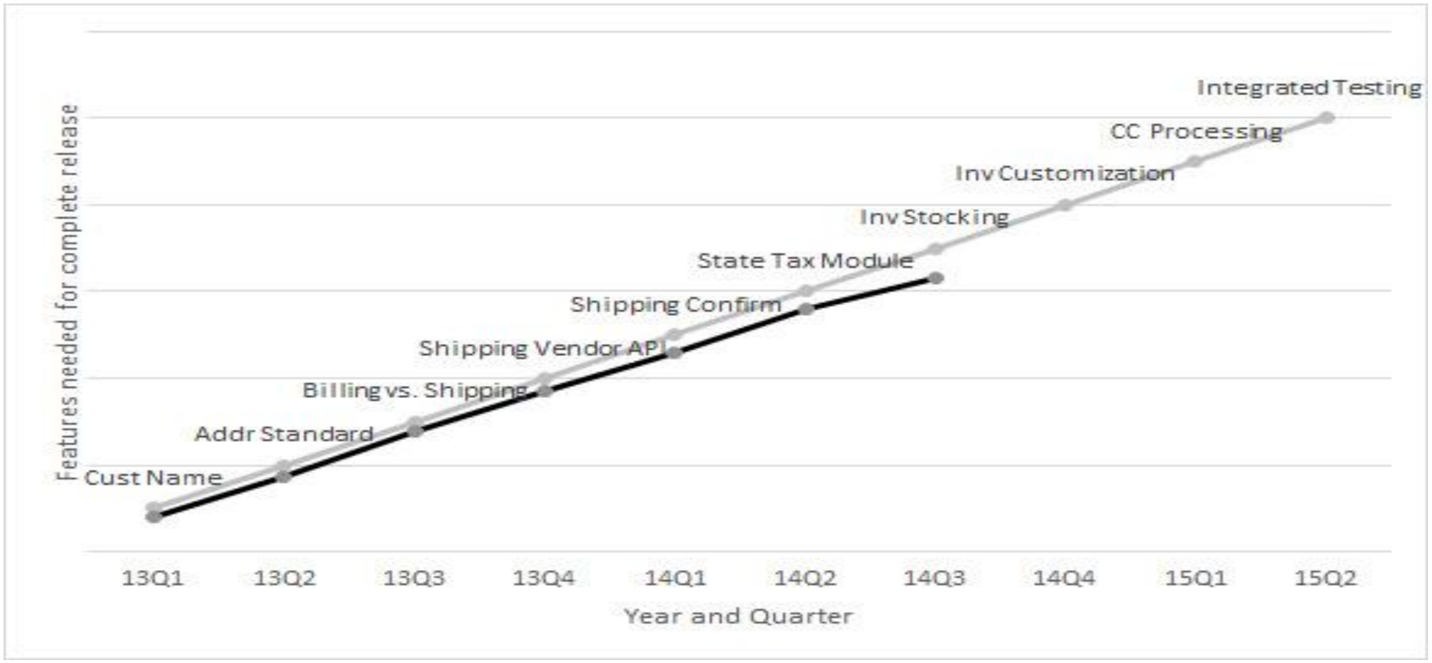
Example burn-up chart by total points



- Here we see the black line as the number of story points that we need to achieve to meet our release goals.

- Our grey “actual” line tracks upward, and we meet the goal after Sprint 10.

Burn-up chart by feature



- This graph shows the feature milestones that we need to complete each quarter to realize the vision for this product.
- The actual progress is then tracked against these features, as completed.

Burn-up Charts

- Although the first two graphs do reveal the progress that the team is making, the charts may not convey the necessary information to stakeholders.

- It is interesting to note the team’s velocity and how quickly they are delivering story points, but that does not necessarily mean that the product is producing the right deliverables.

- The third graph shows the actual features that are delivered to the marketplace. Executives often prefer this view because they can easily see the feature delivery progress.

Burn-down Charts

- Burn-down charts serve a different purpose: These are the daily status checks for the team relative to where they expected to be at a particular point in time.

- This is a critical tracking element within a sprint or iteration to ensure that any necessary course correction is identified and addressed as early as possible.

Burn-down Charts

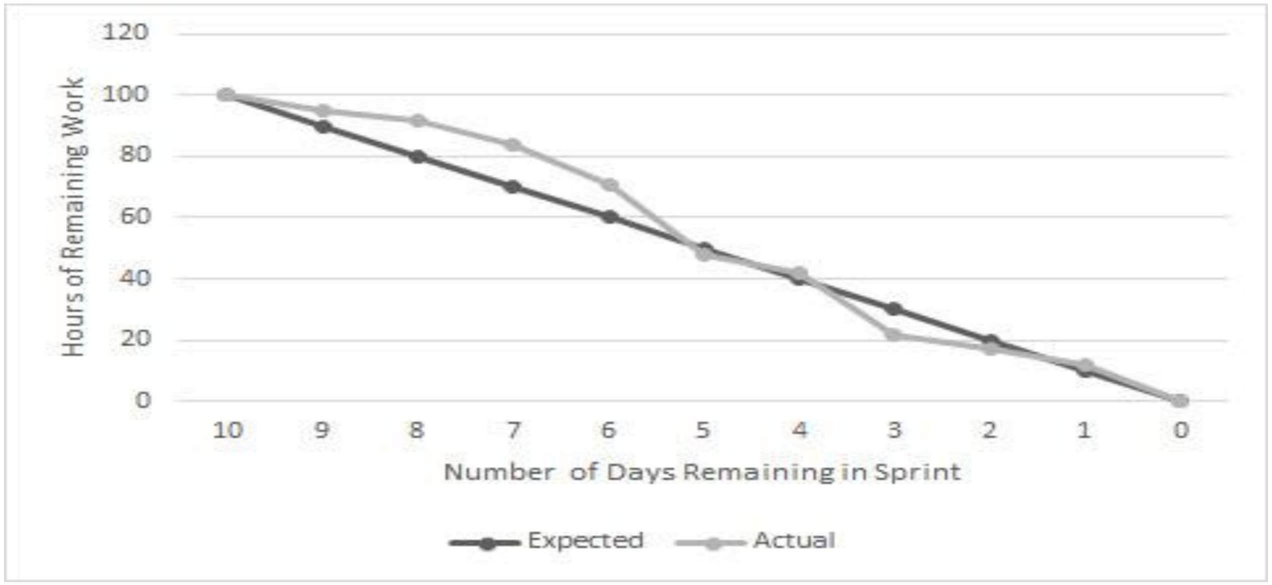
- As previously described, a few critical activities take place during the Sprint planning meeting.

- First, the developers choose the tasks that they want to own. After they select the tasks, they estimate the hours that they believe each task will take them to complete.

Burn-down Charts

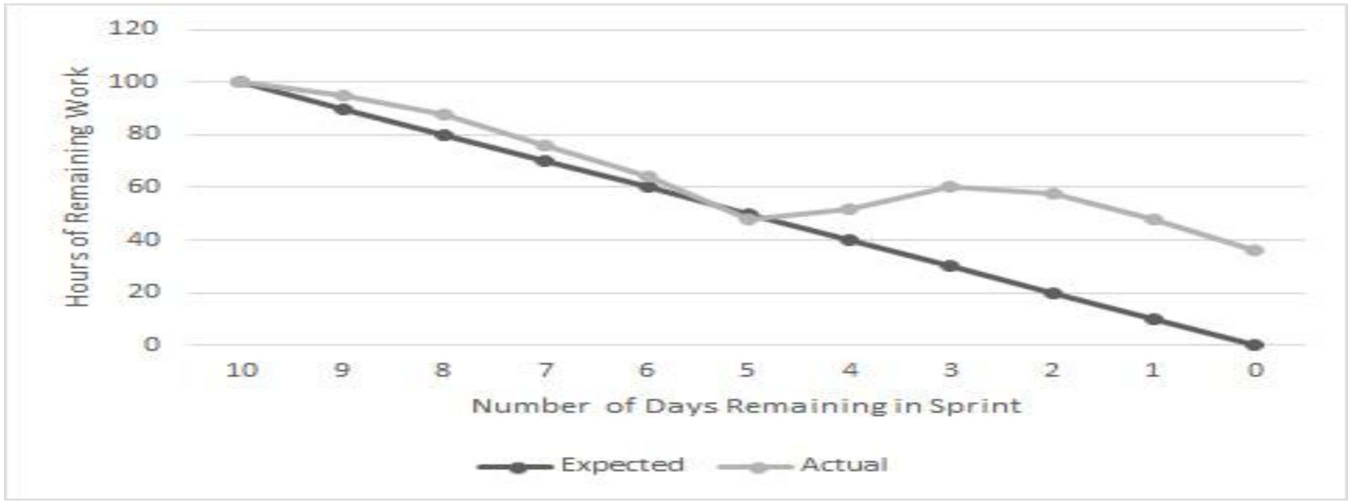
- Before Sprint planning, most activities are at the user story level, and they are estimated in story points.
- Once Sprint planning is completed, stories are broken down into tasks, and tasks are estimated in hours.
- Now that the developer is known, and he or she has small increments of work, it is easier to estimate the actual amount of time a task will take.
- Therefore, the team can see the total number of hours allocated to the sprint and track their actual progress against the target.

Burn-down Charts



The black “expected” line represents the ideal sprint progress if the same level of work was completed every day. You can tell from the graph that the sprint got off of a rocky start because our grey “actual” line was above the “expected” line for several days. Whenever this happens, the team must determine if they are able to finish the expected amount of work within the sprint. In this case, the team recovered nicely, and with three days remaining in the sprint, they were actually ahead of schedule. They finished the last day with all of the work completed.

Burn-down Charts



As you can see from this graph, something went horribly wrong on day 5. Perhaps several team members were out sick with the flu; or the team uncovered a significant problem with the database; or a production issue occurred and the entire development team was pulled off the sprint to work on the live issue; or a story ended up being significantly more difficult than originally thought.

Burn-down Charts

- In any case, this burn-down graph clearly shows that the sprint is in jeopardy, and we can see this almost immediately.
- The team has several options to try to remedy the situation.
- For example, they can reassign tasks to the most competent developers to speed things up, or they can work long hours (with lots of caffeine) to make up the difference (noting that this is not a sustainable way to manage work).
- They can also negotiate with the product owner about the situation and the best resolution.

Burn-down Charts

- The product owner should be involved in all discussions because he or she will need to determine which stories are removed from the sprint, if necessary.

- No one ever wants this to happen, but when it does, having the product owner determine the appropriate work to complete—based on priority and business value—is still far better than not knowing and just having the IT teams fail to deliver.

Burn-down Charts

- Although this example is not ideal, it is still positive. Without Agile and this level of transparency, we would have situations of unknown risk, dates missed, and customers disappointed.

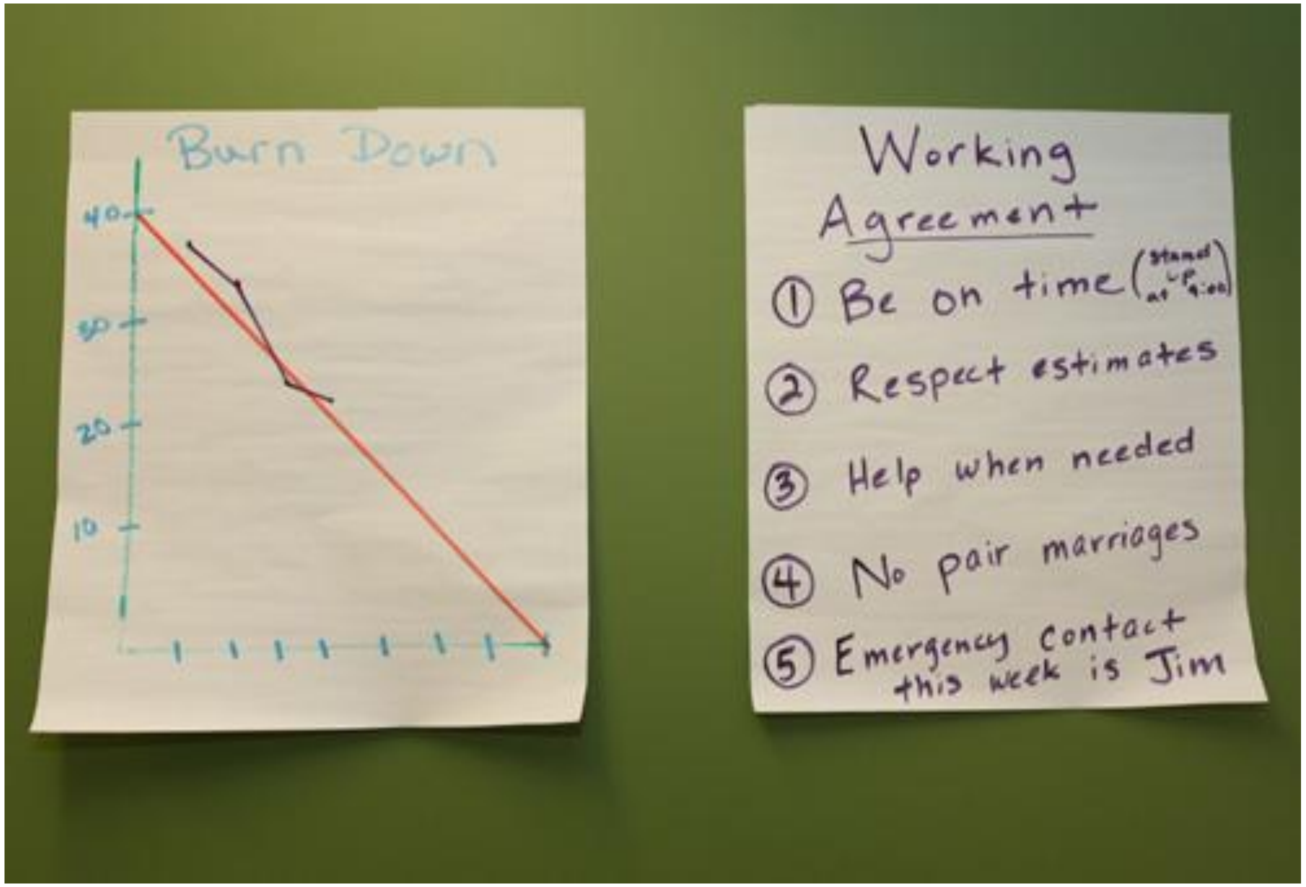
Information Radiators

- **Information radiators** are anything posted in the team members' physical space that they will walk by or see on a regular basis.

- The idea is that the information radiates into the team's subconscious because they encounter the information regularly.

- Radiators work in much the same way as advertisements on billboards along the freeway: They encourage you to think about ideas because they are presented to you regularly.

Information Radiators



Working Agreement

1. Be on time (stand up at 9:00)
2. Respect estimates
3. Help when needed
4. No pair marriages
5. Emergency contact this week is Jim

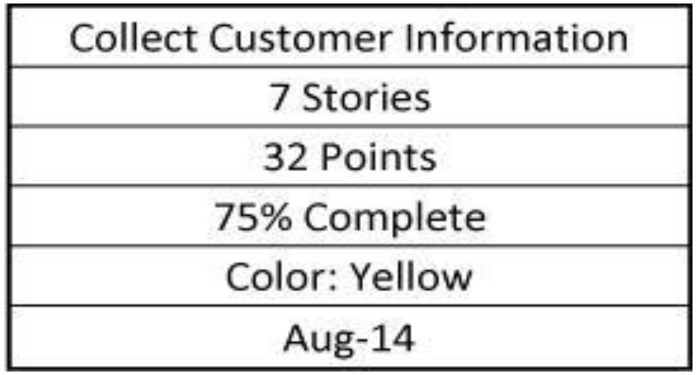
Information Radiators

- For some Agile teams, the information radiator might be some best practices posted on a white board, such as “We will not write new code until all defects are closed.”
- Other teams may post the daily status in a common area so the team members know what is going on with the project and maintain a sense of urgency.
- Some teams may even use information radiators to promote a positive culture and post birthdays or promotions for everyone to see.
- Regardless of how a team decides to use information radiators, they are a helpful tool to promote team communication.

Feature-Driven Development(FDD) Parking Lots

- FDD incorporates an excellent way to track progress on larger projects where many activities are contributing to a cohesive whole.

- E.g. FDD parking lot partial view

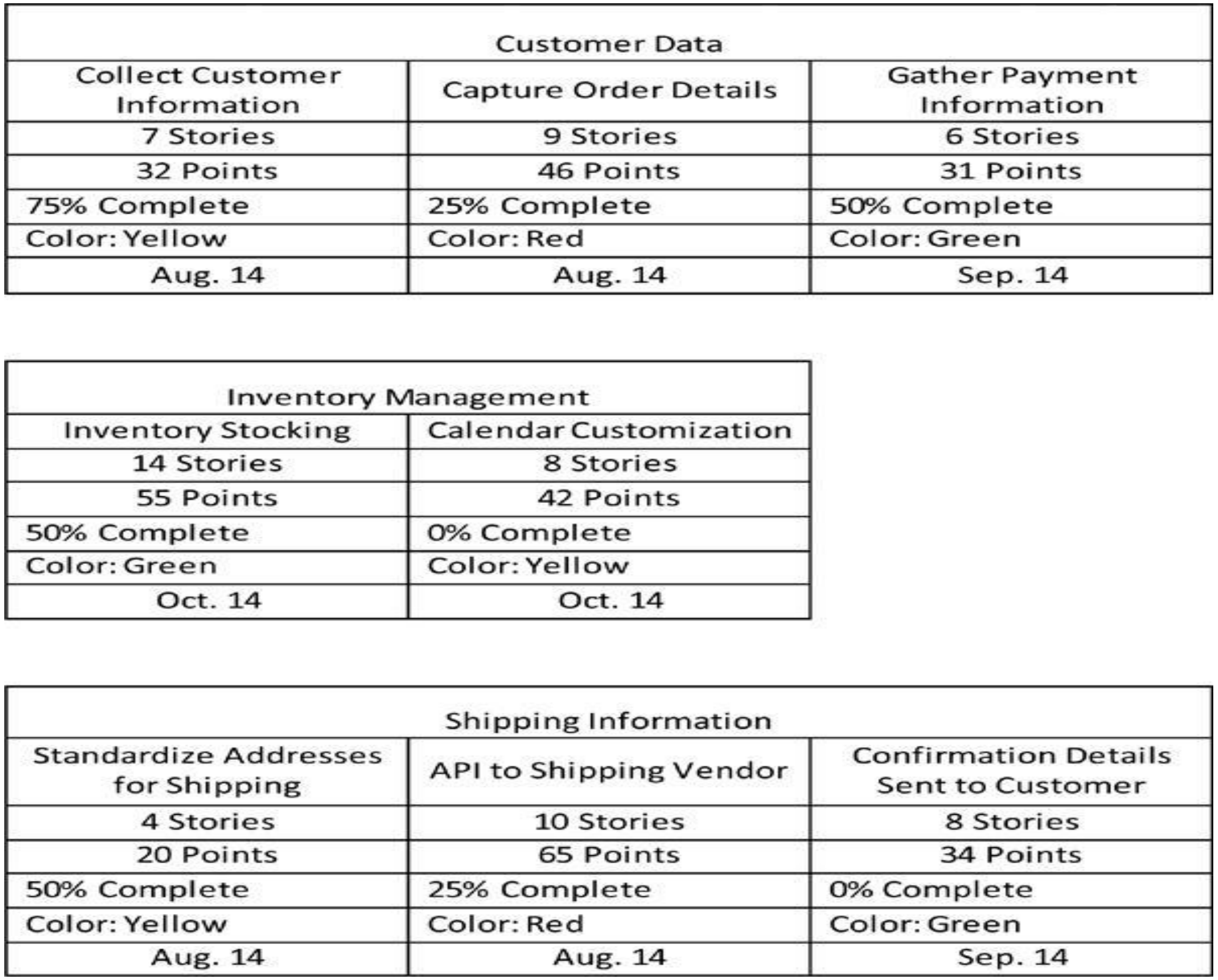


Collect Customer Information
7 Stories
32 Points
75% Complete
Color: Yellow
Aug-14

Feature-Driven Development(FDD) Parking Lots

- This tells us that the feature “Collect Customer Information” consists of seven stories totaling 32 points.
- At this moment, we are 75% complete, and the feature is needed by August 2014.
- The color on the story can indicate its health, this particular story being yellow, meaning it is in jeopardy.
- Although this is an interesting depiction of information, it is not necessarily more valuable than any of the other Agile tools we have discussed—that is, until you add many other components, and then the picture painted by the FDD parking lot is incredibly useful

Feature-Driven Development(FDD) Parking Lots



Customer Data		
Collect Customer Information	Capture Order Details	Gather Payment Information
7 Stories	9 Stories	6 Stories
32 Points	46 Points	31 Points
75% Complete	25% Complete	50% Complete
Color: Yellow	Color: Red	Color: Green
Aug. 14	Aug. 14	Sep. 14

Inventory Management	
Inventory Stocking	Calendar Customization
14 Stories	8 Stories
55 Points	42 Points
50% Complete	0% Complete
Color: Green	Color: Yellow
Oct. 14	Oct. 14

Shipping Information		
Standardize Addresses for Shipping	API to Shipping Vendor	Confirmation Details Sent to Customer
4 Stories	10 Stories	8 Stories
20 Points	65 Points	34 Points
50% Complete	25% Complete	0% Complete
Color: Yellow	Color: Red	Color: Green
Aug. 14	Aug. 14	Sep. 14

Feature-Driven Development(FDD) Parking Lots

- From this parking lot view, you can get an immediate sense for the project health, even though we are looking at a total of 66 stories and 325 points.

- You can add teams/owners to each parking lot and define a more detailed color scheme as needed for your particular project.

- FDD’s model-driven approach allows it to scale to support large, complex projects.

Other Progress Charts

- Gantt (percent complete) and stoplight (red, yellow, or green status) charts have been the most common status-tracking tools in software development projects for many years.

- Although there is nothing about the Agile methodology that would prevent Gantt or stoplight charts from providing useful information, Agile teams have embraced tracking tools that help the teams focus on their daily status.

Tracking Quality

- Agile endorses the idea of writing quality code throughout the development process through paired programming and test-driven development.

- Despite these preventive measures, poor quality can still be introduced into the product.

- There are several approaches to tracking the quality of a product during an iteration.

Tracking Quality – Build Status

- Once a developer has completed testing of his or her code, the first check for quality is to look for errors when the code becomes integrated in the product build on the server.

- If a previously error-free build is indicating that there are errors, then the development and test team have a very clear indication that there may be a problem with the new code.

- These defects are found quickly because Agile teams are doing regular builds. It is also easier to narrow down the defect location in the code because typically only small amounts of code are added with each build.

Tracking Quality – Build Status

- The team can track how many builds introduce errors over time, but this is not commonly measured.

- The Agile teams that utilize a siren or alarm with a build error often create an environment where defects are avoided at all costs and build errors become a rarer occurrence.

Tracking Quality – Defects

- Defects are a normal part of the product development process.
- In Waterfall development, quality is sometimes measured as a ratio such as the number of defects introduced per 1,000 lines of new code or the number of defects detected compared to the number that the developers were able to close in a given week.
- These metrics can also be used while doing Agile development, and in fact some teams create a defect burn-down chart similar to the sprint burn-down chart as a visual representation of the defect closures.

Tracking Quality – Defects

- If the defect burn-down chart is not trending down, then the team knows that either defect closure is not being prioritized by the developers or the new code is continuing to have more and more problems.

- In either case, the team should focus immediately on defect closure.

Tracking Quality – Test Cases

- An important part of the testers' role is to write test cases that will uncover the defects in the code to ensure the highest quality product is released to customers.

- These tests should cover both the functionality and the usability of the product.

- In Waterfall projects, testers would review the design documents and requirements, write an overall test plan, write individual test cases for the requirements, and then execute the test cases when the developers had finished coding and document the results.

- If a defect was found during a test case in Waterfall, it would be documented and prioritized, and the tester would track it through resolution.

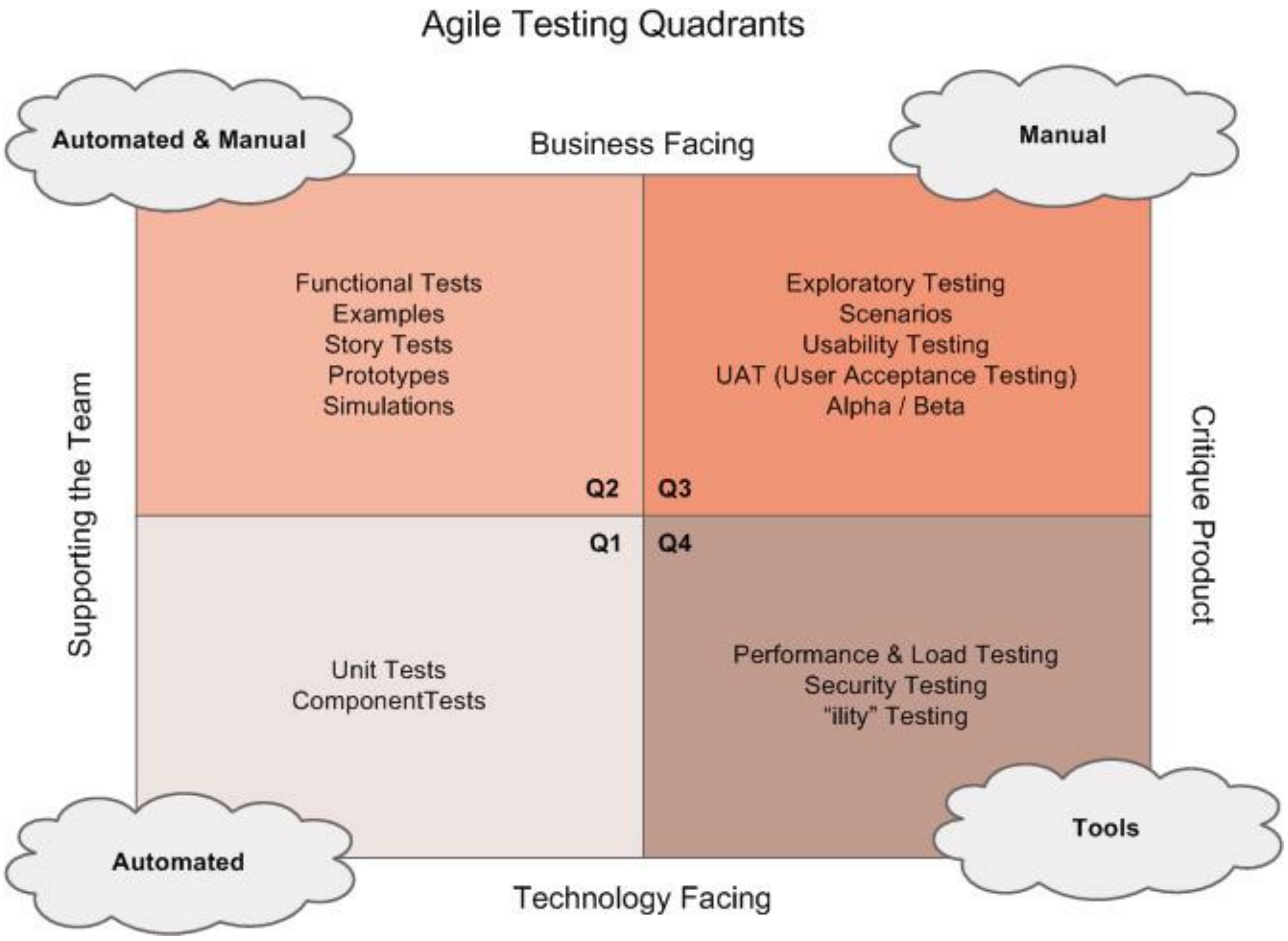
Tracking Quality – Test Cases

- Agile development works differently because testing is considered more of a whole-team activity that happens throughout the iteration.

- Agile testing quadrants describe how an Agile tester can think through the different types of tests that need to be performed during the iteration for the purposes of using test cases to assess the quality of the product.

- It is important to make sure that all aspects of the product (e.g., technical, usability, performance) are sufficiently covered.

Tracking Quality – Test Cases



Tracking Quality – Test Cases

- Savvy Scrum masters know that if very few defects are found during testing, that may mean that the code is of high quality, or it could mean that the product was not sufficiently tested.

- It is important to question the team if too many or too few defects are found during an iteration.

Mocks Objects and Frameworks

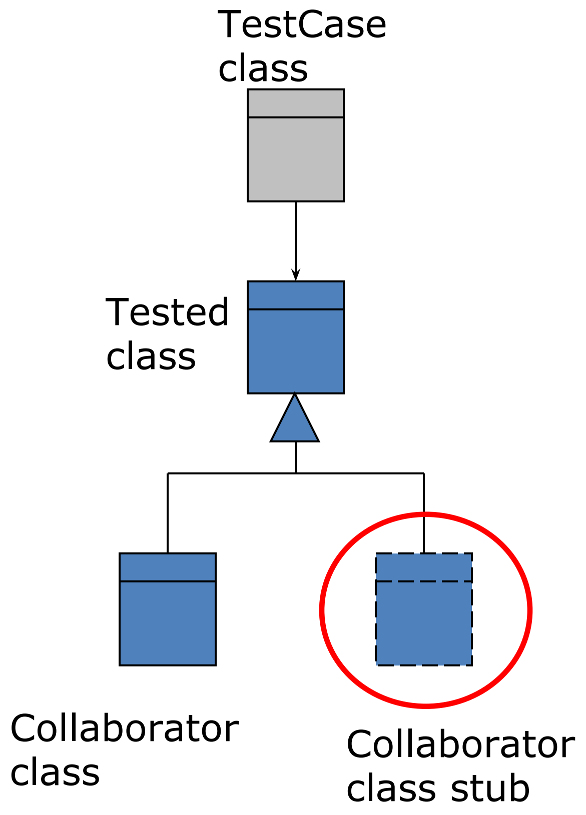
Test Isolation

- Two aspects of test isolation
  - Creating independent tests
    - That don’t depend on other tests.
  - Isolate the unit under test
    - Each (JUnit) test method focuses on a single application method.
    - Each (JUnit) test case class focuses on a single application class.
- But, a the single application class under test may collaborate or interact with other application classes, which may or may not yet have been developed when the application class under test must be tested.

Stubs

- Simulate non-existent classes
- Bare basics implementation
- May support state verifications

• Test early



Stubs

- A simple stub
  - The business logic can be very minimalist

```
Public class Triangle {
  private int a,b,c;

  public Triangle(String params) { }

  protected static boolean validate(String params) {
    return true;
  }

  public double getArea() {
    return 1;
  }
}
```

- Stu

Mock Objects

- The concept behind mock objects is that we want to create an object that will take the place of the real object.

- This mock object will expect a certain method to be called with certain parameters and when that happens, it will return an expected result.

Mock Objects

- This means a Mock object is an object created to stand in for an object that your code collaborates with. Your code can call methods on the mock object, which delivers <u>results</u> as set up by your tests.

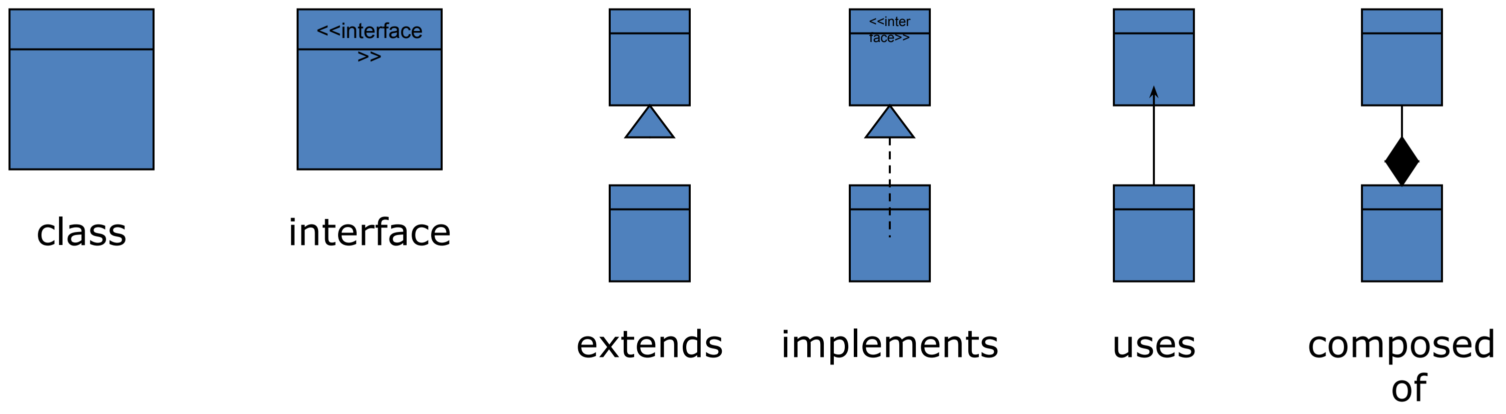
- Suited for testing a portion of code in isolation form the rest of the code.
- Replace the objects with which the methods under test collaborate.
- Do not implement business logic.

Advantages of Mock Objects

- Unit test each method in isolation from other methods or the environment.
- Provides the ability to test code that has not yet been written.
- Allows programming teams to unit-test one part of the code without waiting for the other parts.
- Provides the ability to write focused tests that test only a single method, without the side effects from other objects being called from the method under test.

Mocks and stubs

- Class diagram reminders



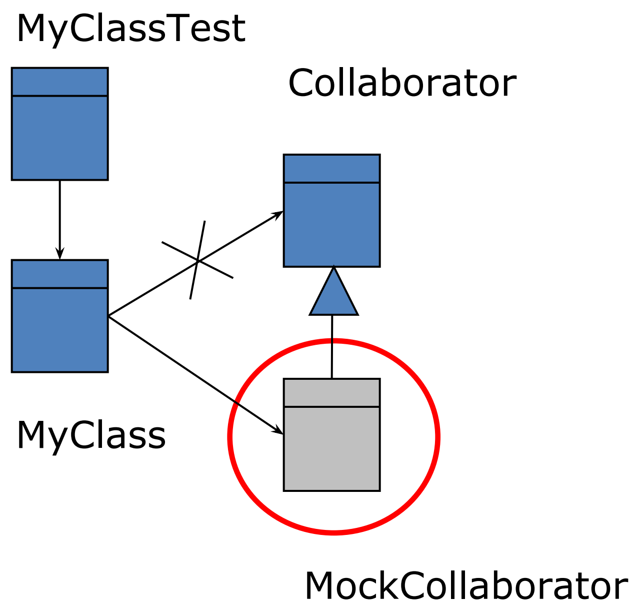
  - Extends: class A inherits from class B
  - Uses: class A uses an instance of class B
  - Implements: class A realizes the characteristics of interface B
  - Composed of: class A includes one or more instances of class B

Static vs. Dynamic Mocks

- Static Mocks
  - Static mock objects are prewritten and hard-coded.
  - They extend the original class, or inherit a common interface.
- Dynamic Mocks
  - Created at runtime.
  - Using a mock framework or tool.
  - Avoids cluttering code with mock classes.

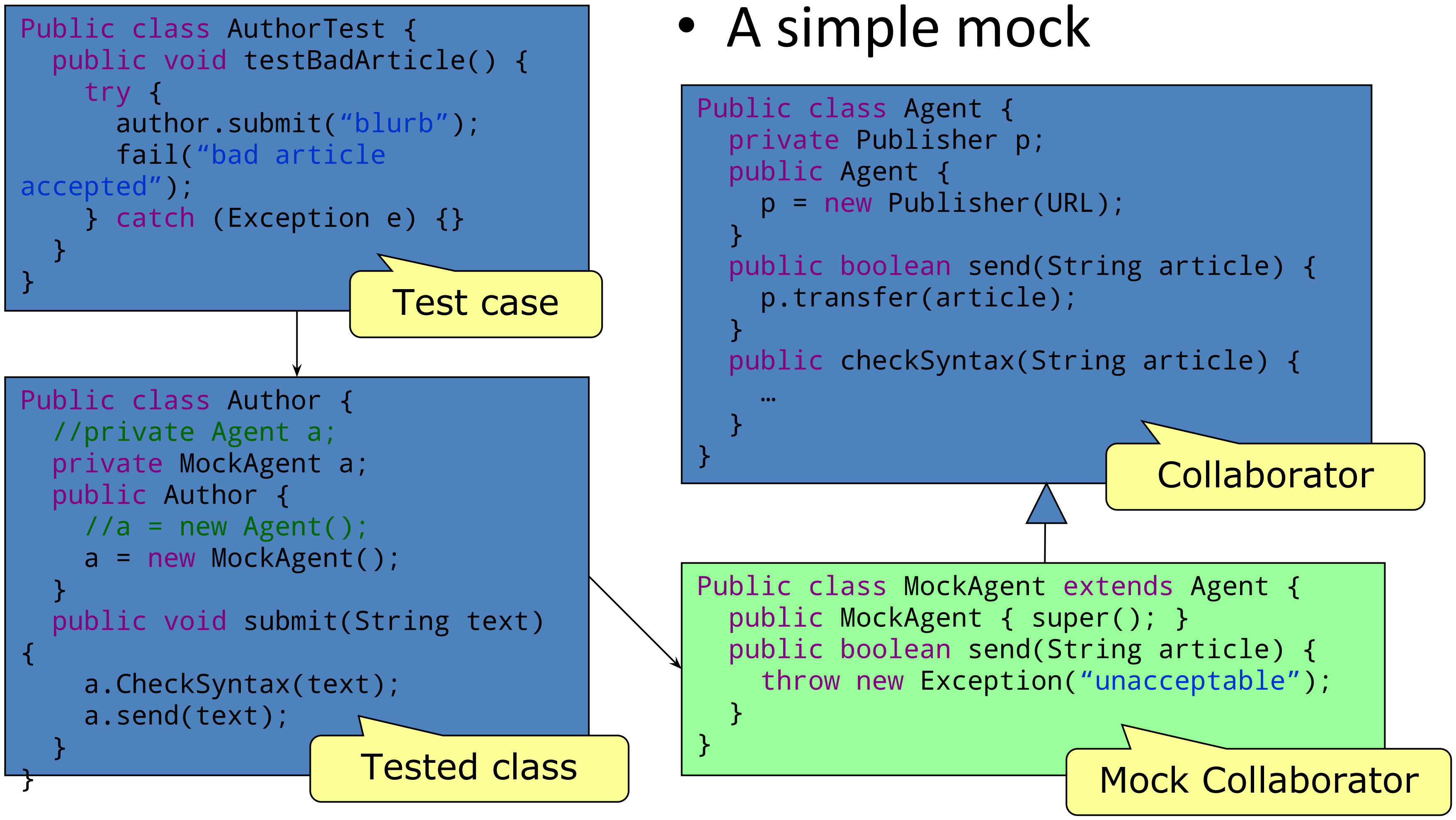
Mocks

- Mocks
  - Simulate existing classes
  - Overrides behaviour
  - Adds test control functionality
  - Usually supports behaviour verification
- Better testing



Static Mocks – A First Example

- A simple mock



```
Public class AuthorTest {
  public void testBadArticle() {
    try {
      author.submit(“blurb”);
      fail(“bad article
accepted”);
    } catch (Exception e) {}
  }
}
```

```
Public class Author {
  //private Agent a;
  private MockAgent a;
  public Author {
    //a = new Agent();
    a = new MockAgent();
  }
  public void submit(String text)
{
    a.CheckSyntax(text);
    a.send(text);
  }
}
```

```
Public class Agent {
  private Publisher p;
  public Agent {
    p = new Publisher(URL);
  }
  public boolean send(String article) {
    p.transfer(article);
  }
  public checkSyntax(String article) {
    …
  }
}
```

```
Public class MockAgent extends Agent {
  public MockAgent { super(); }
  public boolean send(String article) {
    throw new Exception(“unacceptable”);
  }
}
```

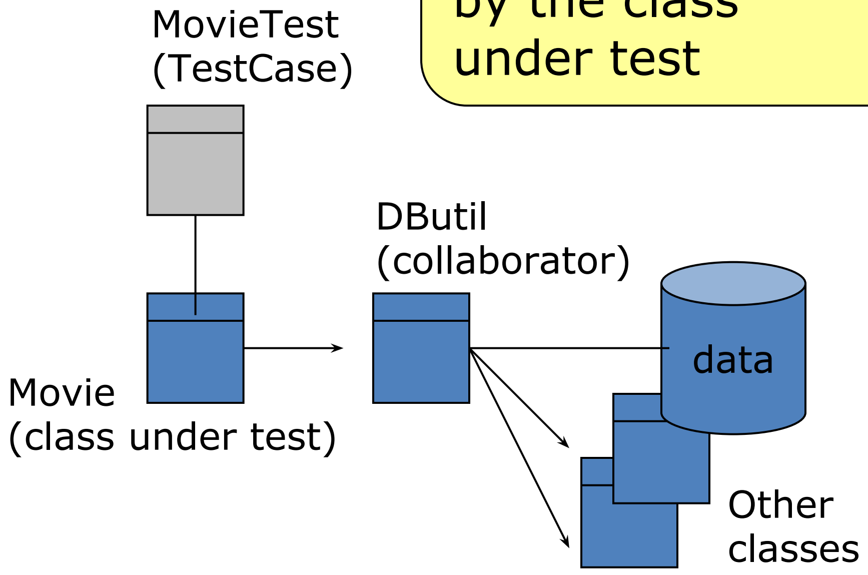
Static Mocks – A First Example

- Problem with this example is that the tested class (product code) must be edited to access the mock Collaborator during test.
- Not best practice.

Static Mocks

- To overcome this problem, Mock Objects can be used as a refactoring technique to ensure that the tested class (product code) is not modified during the unit test process.

Refactoring with Mocks – A better Example



The collaborator class is created by the class under test

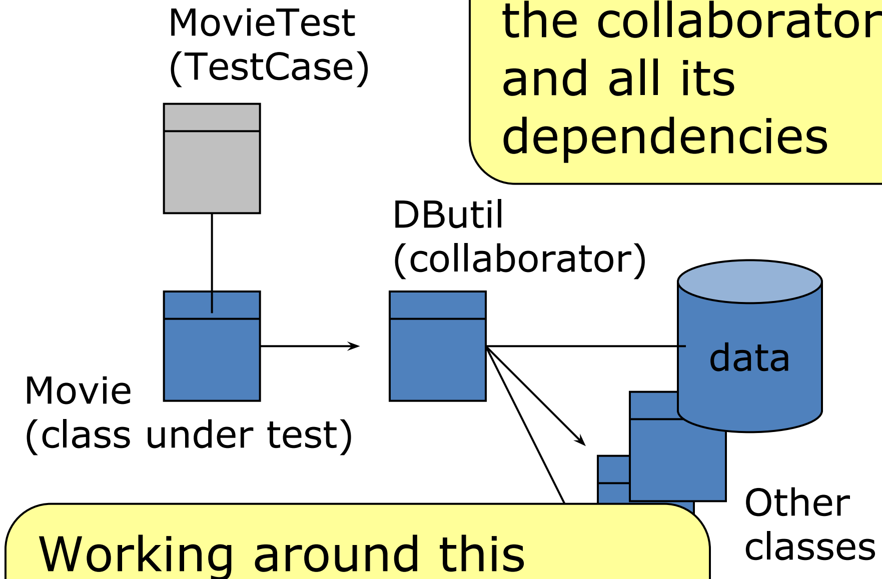
```
public class Movie {
  private String title;
  private DButil dbu;

  public Movie(String title) {
    this.dbu = new Dbutil(URL);
    this.title = title;
  }

  boolean isOn(Date d) {
    return this.dbu
      .run(QUERY, this.title,
d);
  }
}
```

Movie

```
Movie m = new Movie("Titanic");
assertTrue(m.isOn(TODAY));
```

MovieTest assertion

Refactoring with Mocks – A better Example



The test involves the collaborator and all its dependencies

Working around this problem involves changes to the class or collaborator code - bad practice

```
public class Movie {
  private String title;
  private DButil dbu;

  public Movie(String title) {
    this.dbu = new Dbutil(URL);
    this.title = title;
  }

  boolean isOn(Date d) {
    return this.dbu
      .run(QUERY, this.title,
d);
  }
}
```

Movie

```
Movie m = new Movie("Titanic");
assertTrue(m.isOn(TODAY));
```

MovieTest assertion

Refactoring with Mocks – A better Example

- Suggests changing the application code.
- For this example, employ the Inversion Of Control (IoC) design pattern to refactor the application code (i.e. Movie class and DBUtil class), which isolates classes form their collaborators.

Refactoring with Mocks – A better Example

- Inversion Of Control – definition

  "Applying the IoC pattern to a class means removing the creation of all object instances for which this class is not directly responsible and passing any needed instances instead. The instances may be passed using a specific constructor, using a setter, or as parameters of the methods needing them. It becomes the responsibility of the calling code to correctly set the domain objects on the called class"

- a.k.a. Dependency Injection

Refactoring with Mocks – A better Example

- Removing the dependency from Movie

```
public class Movie {
  private String title;
  private DButil dbu;

  public Movie(String title) {
    this.title = title;
  }

  public setDBUtil(DBUtil dbu) {
    this.dbu = dbu;
  }

  boolean isOn(Date d) {
    return this.dbu
      .run(QUERY, this.title,
d);
  }
}
```

Movie

IoC allows the class user more control over the dependencies. (hence the name "dependency injection")

```
Movie m = new Movie("Titanic");
DBUtil dbu = new DBUtil(URL);
m.setDBUtil(dbu);

assertTrue(m.isOn(TODAY));
```

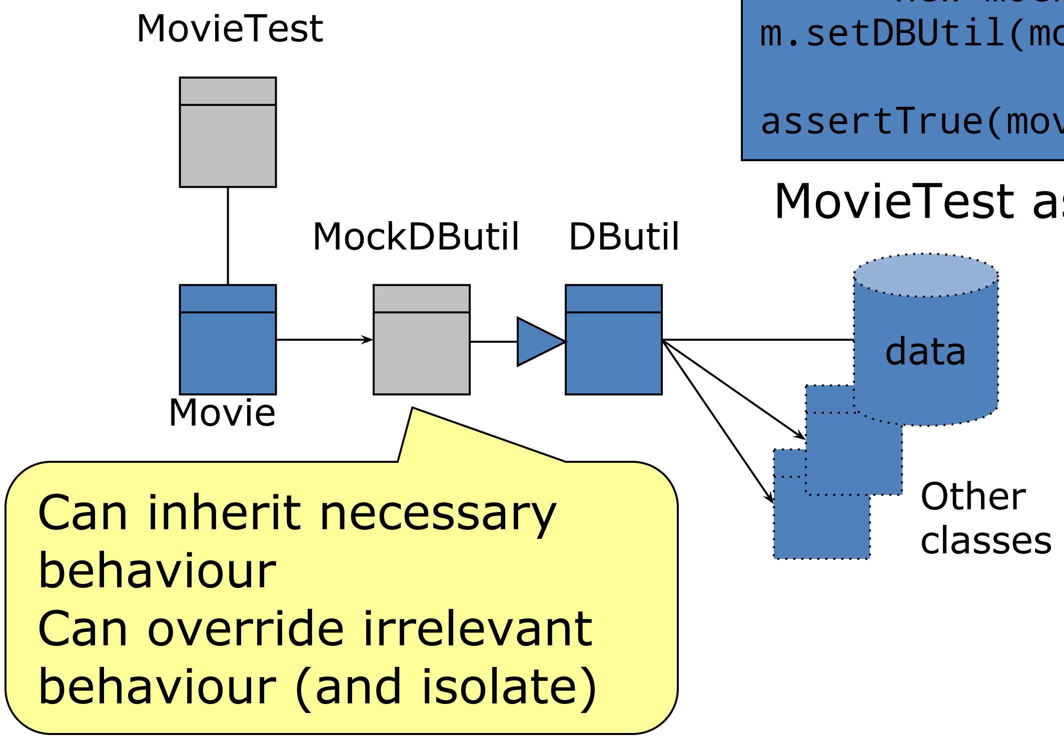
MovieTest assertion

Refactoring with Mocks – A better Example

- Employing a mock object is now much easier

```
Movie m = new Movie("Titanic");
MockDBUtil mockDbu
    = new MockDBUtil(URL);
m.setDBUtil(mockDbu);

assertTrue(movie.isOn(TODAY));
```

MovieTest assertion


MovieTest
MockDButil
DButil
data
Movie
Other classes
Can inherit necessary behaviour
Can override irrelevant behaviour (and isolate)

IoC allows the unit test to use a mock without changing the code of the tested class

Thoughts on Exceptions and Mocks

- Are exceptions collaborator classes?
  - Arguably, yes
  - But no need to worry about isolation
    - Exception classes are just message place-holders with no real business logic.
    - Using mock exceptions would break the exception handling mechanism.

Dynamic Mocks

Mock Frameworks

- There are essentially two main types of mock object frameworks:
  - ones that are implemented via proxy and
  - ones that are implemented via class remapping.
- Let's take a look at the first (and by far more popular) option, proxy.

Mock Frameworks – Proxy Object

- A proxy object is an object that is used to take the place of a real object.

- In the case of mock objects, a proxy object is used to imitate the real object your code is dependent on.

- You create a proxy object with the mocking framework, and then set it on the object using either a setter or constructor.

Mock Frameworks – Proxy Object

- This points out an inherent issue with mocking using proxy objects.

- You have to be able to set the dependency up through an external means – see previous refactoring example.

- In other words, you can't create the dependency by calling ***new MyObject()*** since there is no way to mock that with a proxy object.

Proxy Pattern

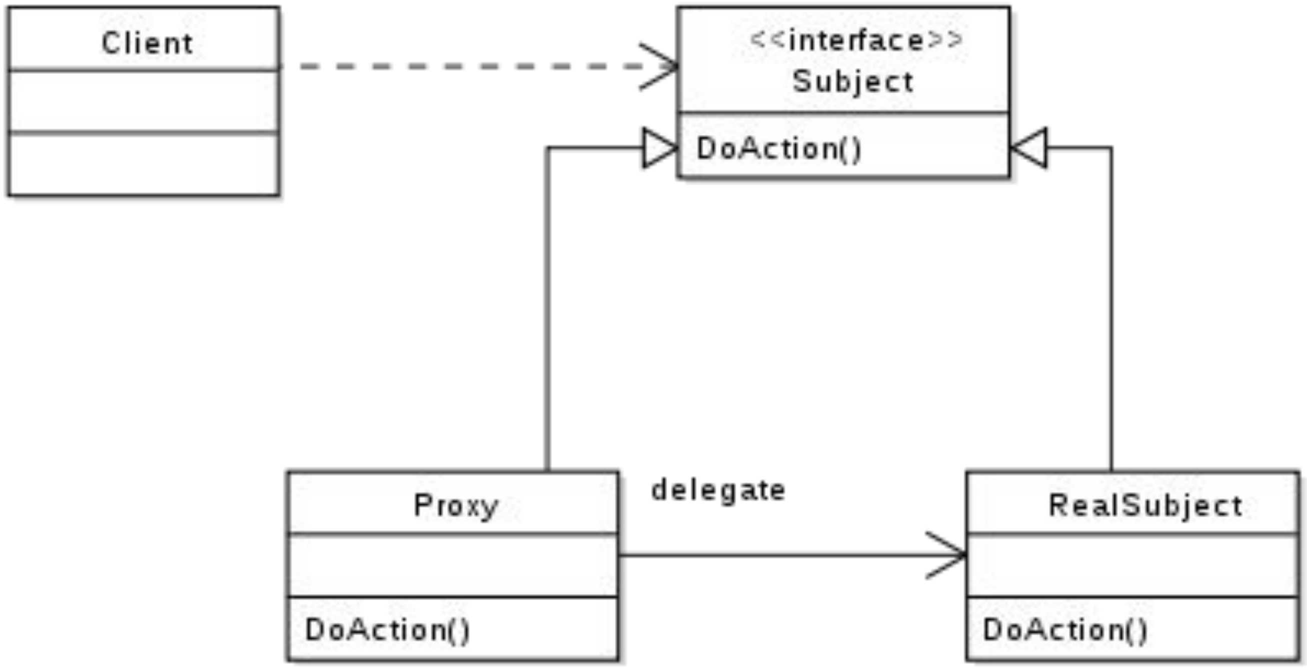
- In computer programming the **proxy pattern** is a software design pattern.
- A *proxy*, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate.

Proxy Pattern

- In short, a proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes.
- In the proxy extra functionality can be provided, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked.

- For the client, usage of a proxy object is similar to using the real object, because both implement the same interface.

Proxy Pattern



What is a class loader?

- The class loader concept, one of the cornerstones of the Java virtual machine, describes the behavior of converting a named class into the bits responsible for implementing that class.

- Because class loaders exist, the Java run time does not need to know anything about files and file systems when running Java programs.

- http://www.javaworld.com/article/2077260/learn-java/learn-java-the-basics-of-java-class-loaders.html

Mock Frameworks – Class Remapping

- The second form of mocking is to remap the class file in the class loader.

- What happens is that you tell the class loader to remap the reference to the class file it will load.

Mock Frameworks – Class Remapping

- So let's say that I have a class MyDependency with the corresponding .class file called MyDependency.class and I want to mock it to use MyMock instead.

- By using this type of mock objects, you will actually remap in the classloader the reference from MyDependency to MyMock.class.

- This allows you to be able to mock objects that are created by using the new operator.

Mock Frameworks – Class Remapping

- Although this approach provides more power than the proxy object approach, it is also harder/more confusing to get going given the knowledge of classloaders you need to really be able to use all its features.

EasyMock – A Mock Framework

EasyMock

- EasyMock is a mocking framework, JAVA-based library that is used for effective unit testing of JAVA applications.

- EasyMock is used to mock interfaces so that a dummy functionality can be added to a mock interface that can be used in unit testing.

EasyMock

- EasyMock facilitates creating mock objects seamlessly.

- The Mock objects are nothing but proxy for actual implementations.

Benefits of EasyMock

- **No Handwriting** – No need to write mock objects on your own.
- **Refactoring Safe** – Renaming interface method names or reordering parameters will not break the test code as Mocks are created at runtime.
- **Return value support** – Supports return values.
- **Exception support** – Supports exceptions.
- **Order check support** – Supports check on order of method calls.
- **Annotation support** – Supports creating mocks using annotation.

EasyMock Setup – Example Steps

1. In Eclipse, create a new java project and call it JUnitTutorial.

2. Right click on your new project and select New --> Folder.

3. Name it lib and click Finish.
4. Usually you don't want to package your test code with your regular code, so let's make an additional source directory, test.

EasyMock Setup – Example Steps

5. To do that, right click on your new project and select Properties.
6. Select Java Build Path from the available options.

7. In the Java Build Path window, click Add Folder.
8. From the Add Folder dialog, select Create New Folder, name it test and click Finish.

EasyMock Setup – Example Steps

9. We will also need to download and add the EasyMock jar files to our project.
10. Once you download the zip file extract the easymock.jar file and place it in the lib folder you created earlier.
11. In Eclipse, right click on your project and select Properties.

EasyMock Setup – Example Steps

12. On the menu to the left, click Java Build Path and select the Libraries tab. Click the button Add External Jars on the right.
13. In the window that pops up, add the easymock.jar and click Ok.
14. Click Ok to close the Properties window.
15. You should now be ready to start your development.

EasyMock – Example

- See attached Word Document on Moodle.

EasyMock Tutorials

- http://www.tutorialspoint.com/easymock/easymock_junit_integration.htm

State Transition Testing

System State

- A system state defines the current condition of a system at a given time.
- For a given state the system will:
  - Process certain inputs only
  - Allow the system state to change to certain other system states.

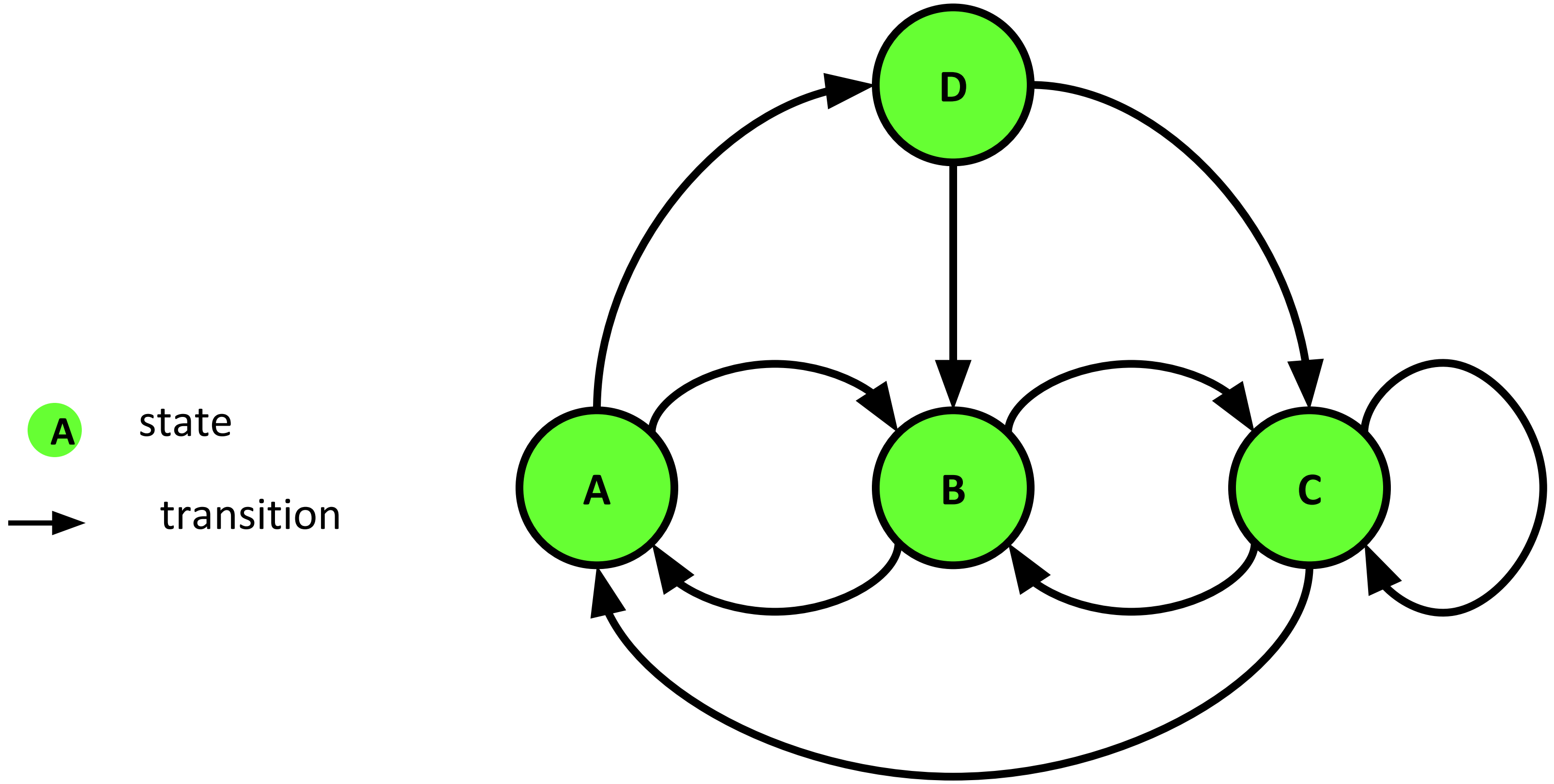
A State Transition

- A State Transition consists of an:

1. Initial state
2. Input that causes the transition to next/final state
3. Next/final state (may be same as initial state)
4. Output that accompanies the transition to next/final state

State transition diagram

State transition diagrams are useful for modelling the behaviour of (state – oriented) systems.



State Diagrams – Construction Rules

- States represented by circular or rectangular shapes.

- Transitions are directed lines labelled with input and output.

- “Dead States” not allowed.

- “Unreachable States” not allowed.

State Transition Testing

- State transition testing is appropriate where the behaviour of a system is described as a set of states and transitions.

- Testing determines if the system changes state correctly.

- State transition tests (for valid transitions) are designed by:
  1. drawing the state-diagram of the system under test,
  2. Convert the diagram to a state transition table based on desired coverage target, and
  3. derive test data for the state transitions.

State Transition - Chow's coverage

- Refer to BS7925-2 for details.
- 0-switch
  - All transitions (1 path, 2 states)
- 1-switch
  - All two-transitions paths (2 paths, 3 states)
- 2-switch
  - All three transitions paths (3 paths, 4 states)
- Etc .....
- A limitation of the test cases derived to achieve switch coverage is that they are designed to exercise only the valid transitions in the component.

State Transition Testing

- A State Table explicitly shows both valid and invalid transitions.

- The State Table is then converted in a State Transition Table.

- Any entry where the state remains the same and the output is shown as null (N) represents a null transition, where any *actual transition that can be induced will represent a failure. It is the testing of* these null transitions that is ignored by test sets designed just to achieve switch coverage.

State Transition Example – BS 7925-2

Consider a component, *manage_display_changes,* with the following specification*:*

The component responds to input requests to change an externally held display mode for a time display device. The external display mode can be set to one of four values: Two correspond to displaying either the time or the date, and the other two correspond to modes used when altering either the time or date. There are four possible input requests: 'Change Mode', 'Reset', 'Time Set' and 'Date Set'. A 'Change Mode' input request shall cause the display mode to move between the 'display time' and 'display date' values. If the display mode is set to 'display time' or 'display date' then a 'Reset' input request shall cause the display mode to be set to the corresponding 'alter time' or 'alter date' modes. The 'Time Set' input request shall cause the display mode to return to 'display time' from 'alter time' while similarly the 'Date Set' input request shall cause the display mode to return to 'display date' from 'alter date'.

State Transition Example – BS 7925-2

- On the white board the tutor will
  1. Draw the State Transition Diagram,
  2. Draw the State Transition Tables for
     - 0 – switch coverage
     - 1 – switch coverage
  3. Draw State Table (for valid & invalid transitions)
  4. Draw the State Transition Table for 3 above.

Decision Table Testing

When to Use Decision Tables

- A Decision Table has two parts:
  - In the Upper Half, the conditions (inputs) are listed,
  - In the Lower Half, the actions (outputs) are listed.

- Every column in the Decision Table is a Test Case.

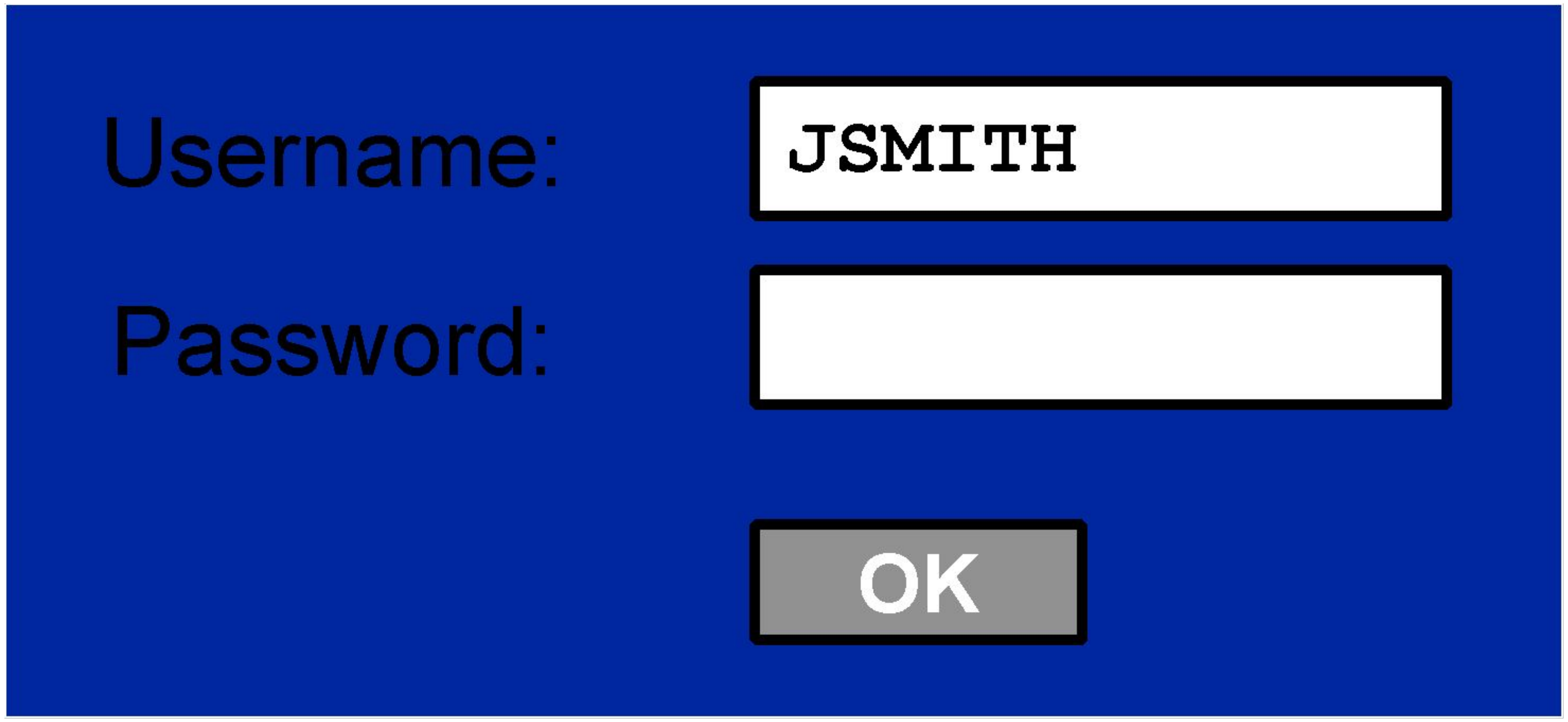
- The objective of testing using Decision Tables is to design tests for executing “interesting” combinations of conditions. Interesting in the sense that possible failures can be detected.

When to Use Decision Tables

- In the least optimised case, every combination of conditions is considered a test case.
- However, conditions may influence or exclude each other in a way that all combinations do not make sense. For combinations that do not make sense, Decision Table Reduction or Optimisation can be employed.

- In the simplest form of the decision table (bivariate table), the fulfilment of every condition and action is noted with a “YES” or “NO”. In an extended table entry, there may be many alternatives for each condition and action.
- For bivariate decision tables, each condition and action should occur at least once with a “YES” and “NO” in the table.

A simple/sample requirement



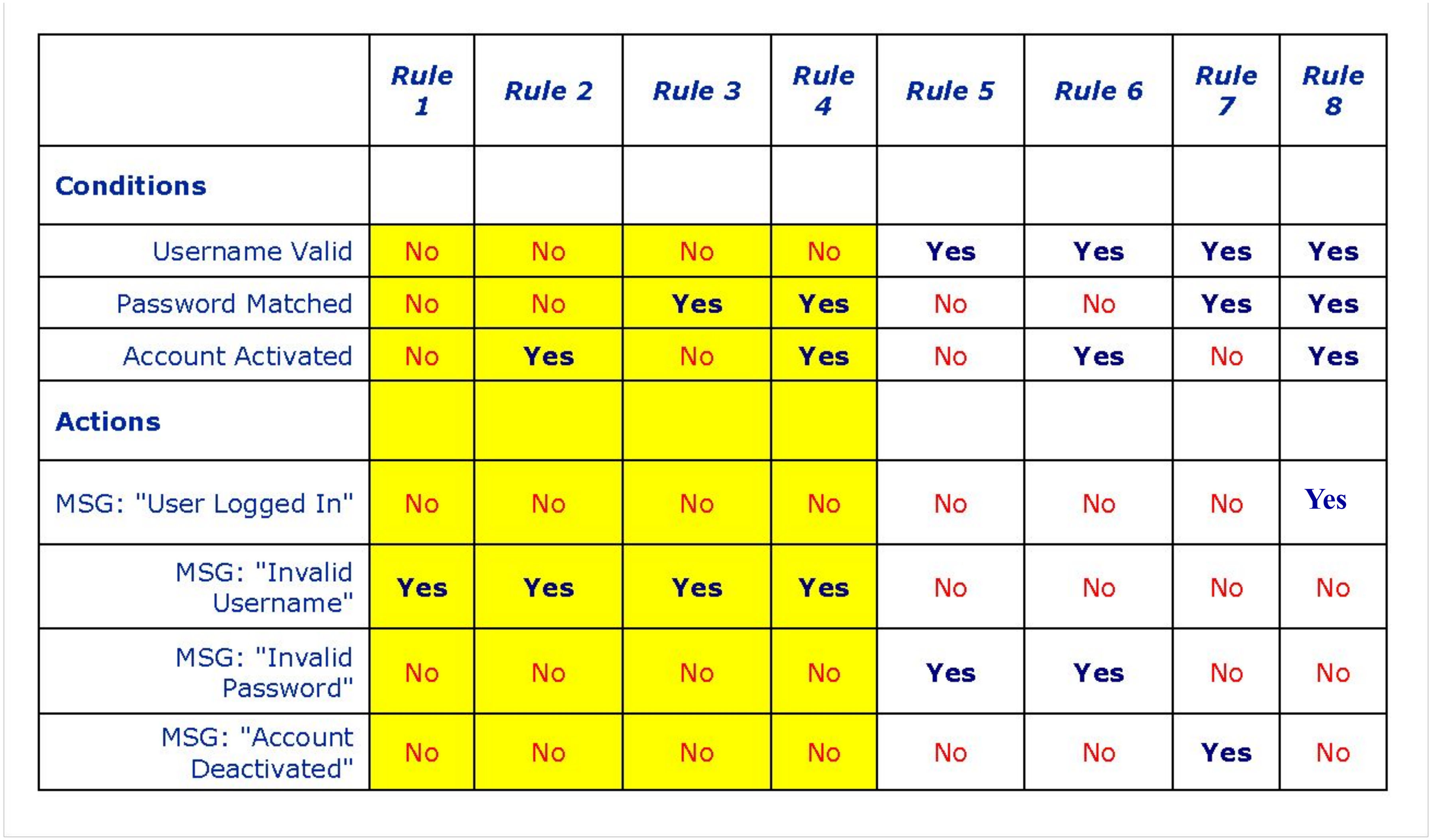
Username: JSMITH

Password:

Sample requirement - log in screen

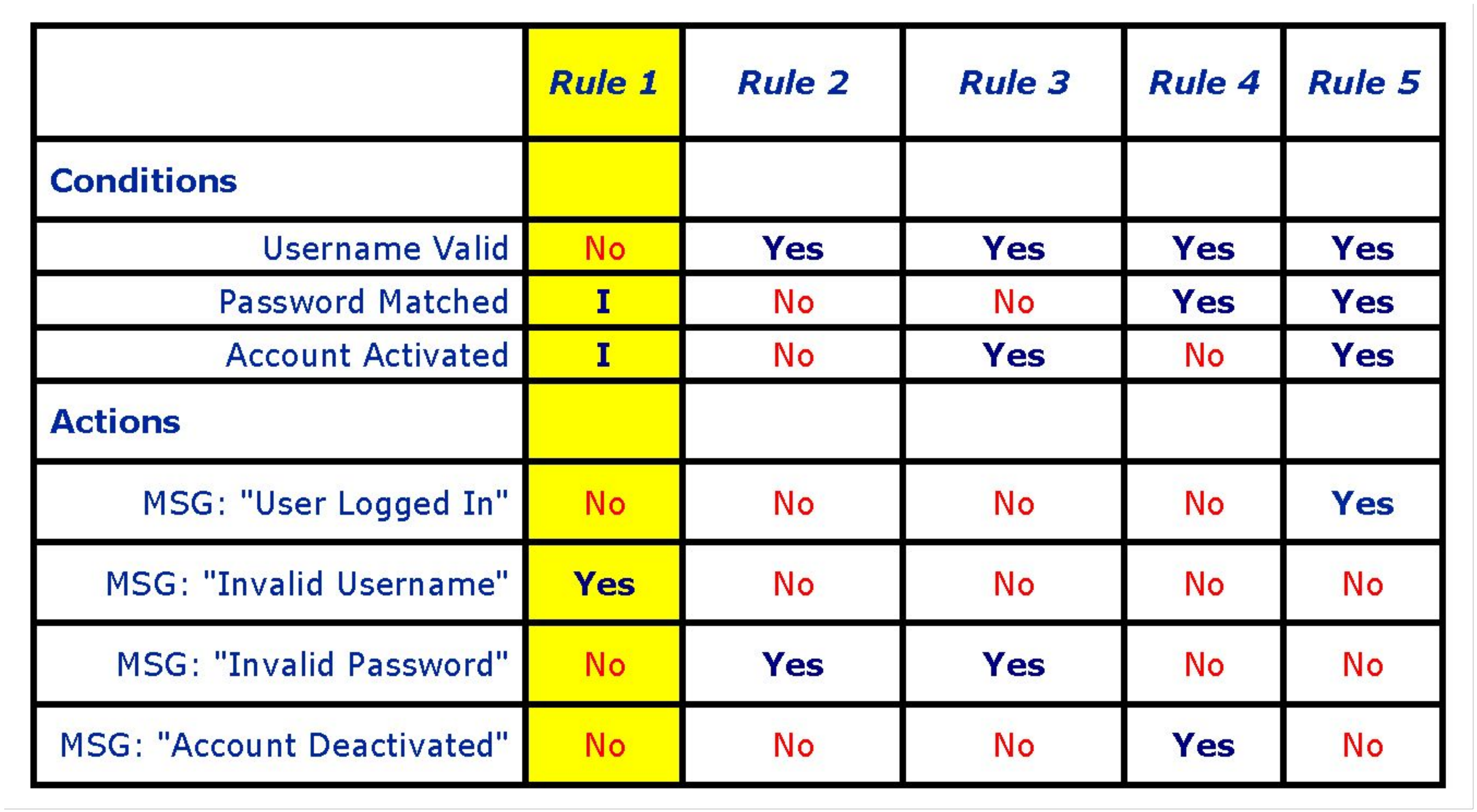
	Rule 1	*Rule 2*	*Rule 3*	*Rule 4*	*Rule 5*	*Rule 6*	*Rule 7*	*Rule 8*
Conditions								
Username Valid	**No**	**No**	**No**	**No**	**Yes**	**Yes**	**Yes**	**Yes**
Password Matched	**No**	**No**	**Yes**	**Yes**	**No**	**No**	**Yes**	**Yes**
Account Activated	**No**	**Yes**	**No**	**Yes**	**No**	**Yes**	**No**	**Yes**
Actions								
MSG: "User Logged In"	**No**	**No**	**No**	**No**	**No**	**No**	**No**	**Yes**
MSG: "Invalid Username"	**Yes**	**Yes**	**Yes**	**Yes**	**No**	**No**	**No**	**No**
MSG: "Invalid Password"	**No**	**No**	**No**	**No**	**Yes**	**Yes**	**No**	**No**
MSG: "Account Deactivated"	**No**	**No**	**No**	**No**	**No**	**No**	**Yes**	**No**

Sample requirement - login screen



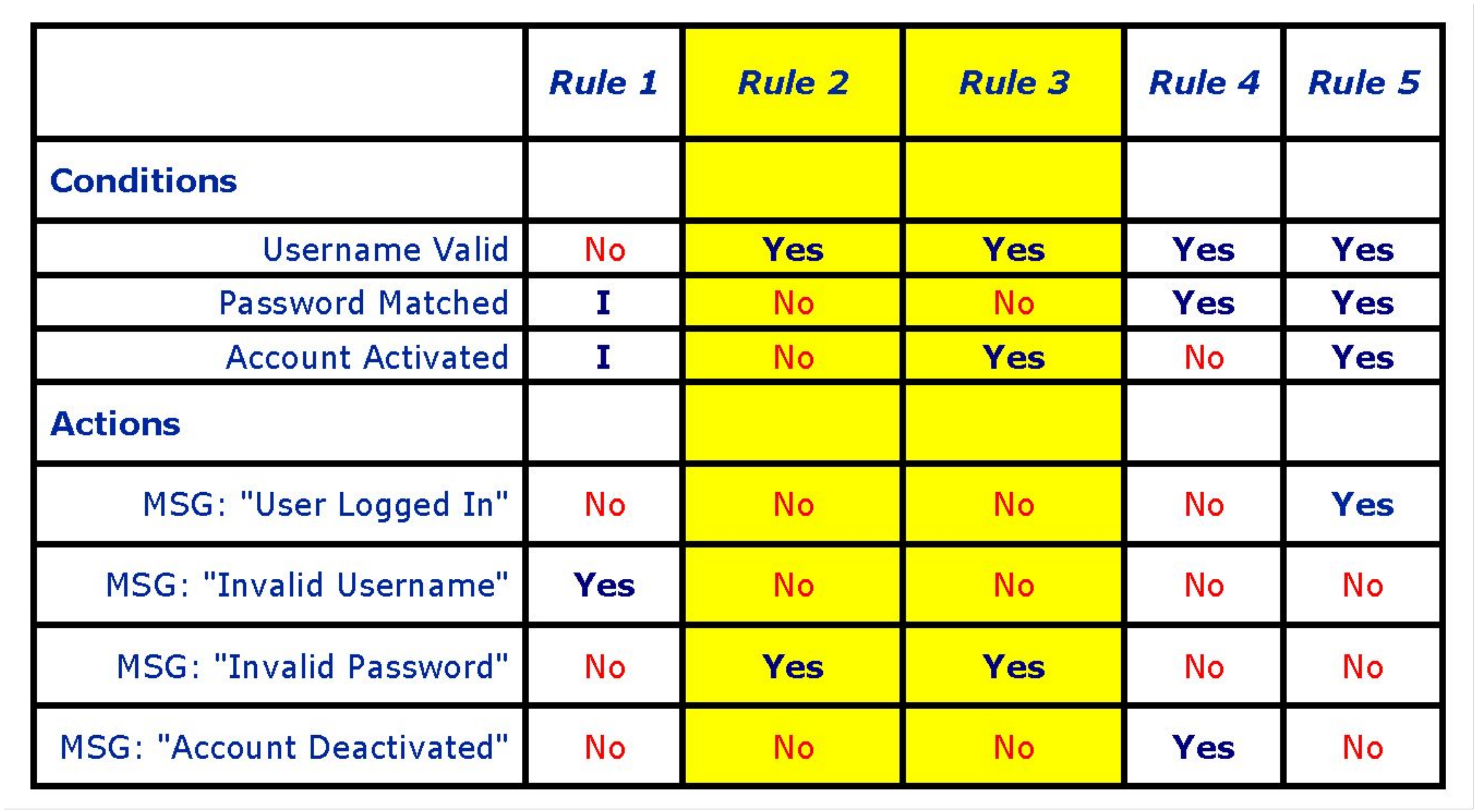
	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
Conditions								
Username Valid	No	No	No	No	Yes	Yes	Yes	Yes
Password Matched	No	No	Yes	Yes	No	No	Yes	Yes
Account Activated	No	Yes	No	Yes	No	Yes	No	Yes
Actions								
MSG: "User Logged In"	No	No	No	No	No	No	No	Yes
MSG: "Invalid Username"	Yes	Yes	Yes	Yes	No	No	No	No
MSG: "Invalid Password"	No	No	No	No	Yes	Yes	No	No
MSG: "Account Deactivated"	No	No	No	No	No	No	Yes	No

Decision table - first reduction



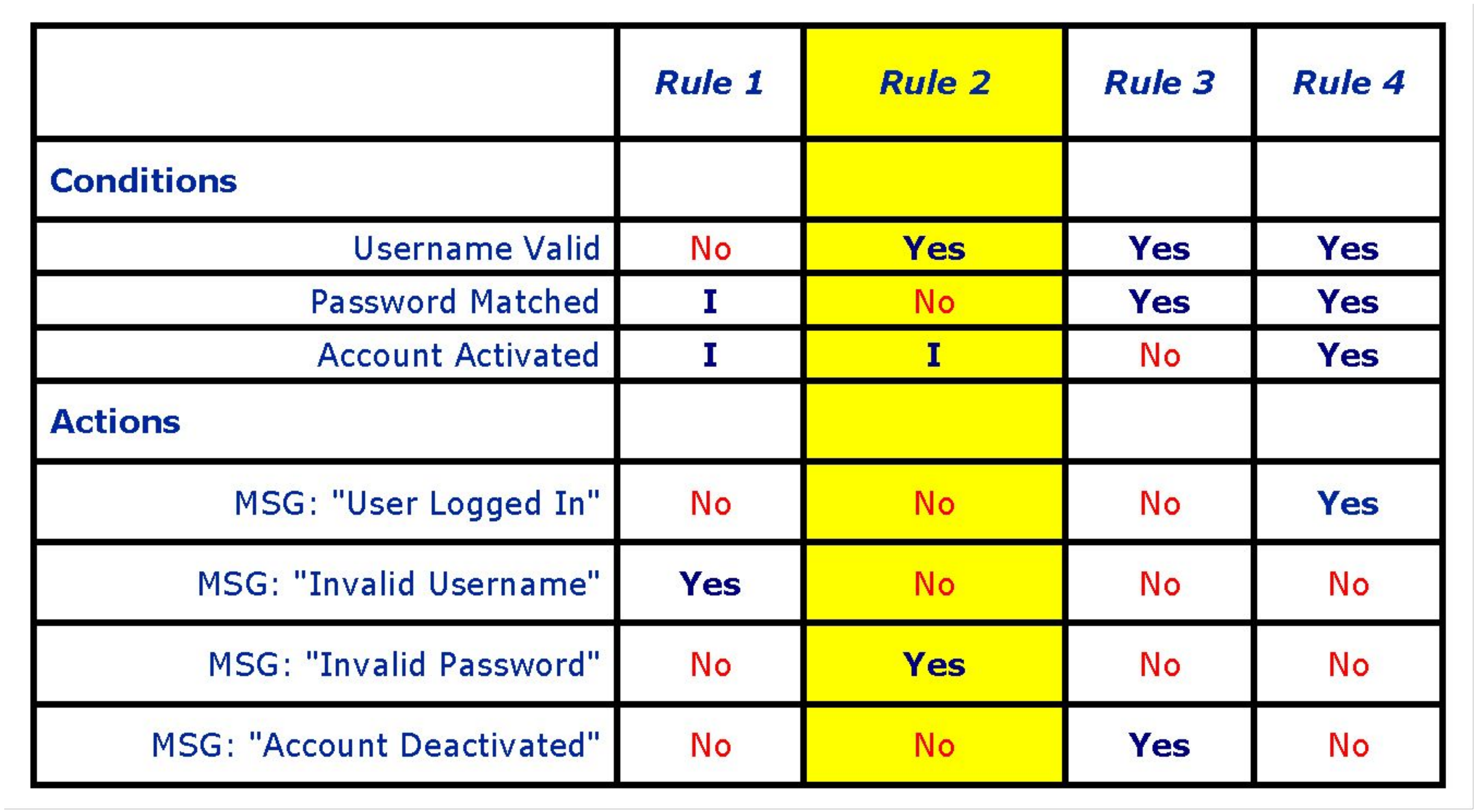
	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5
Conditions					
Username Valid	No	Yes	Yes	Yes	Yes
Password Matched	I	No	No	Yes	Yes
Account Activated	I	No	Yes	No	Yes
Actions					
MSG: "User Logged In"	No	No	No	No	Yes
MSG: "Invalid Username"	Yes	No	No	No	No
MSG: "Invalid Password"	No	Yes	Yes	No	No
MSG: "Account Deactivated"	No	No	No	Yes	No

Decision table - review



	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5
Conditions					
Username Valid	No	Yes	Yes	Yes	Yes
Password Matched	I	No	No	Yes	Yes
Account Activated	I	No	Yes	No	Yes
Actions					
MSG: "User Logged In"	No	No	No	No	Yes
MSG: "Invalid Username"	Yes	No	No	No	No
MSG: "Invalid Password"	No	Yes	Yes	No	No
MSG: "Account Deactivated"	No	No	No	Yes	No

Decision table - second reduction



	Rule 1	Rule 2	Rule 3	Rule 4
Conditions				
Username Valid	No	Yes	Yes	Yes
Password Matched	I	No	Yes	Yes
Account Activated	I	I	No	Yes
Actions				
MSG: "User Logged In"	No	No	No	Yes
MSG: "Invalid Username"	Yes	No	No	No
MSG: "Invalid Password"	No	Yes	No	No
MSG: "Account Deactivated"	No	No	Yes	No

Decision table testing - Guidance

- Requirements often include many conditions and actions.
- Requirements often have default or catch-all rules
- Use boundary values to increase yield

Developing Decision tables

In order to build decision tables, you need to:

- determine the maximum size of the table,
- eliminate any impossible situations,
- eliminate any inconsistencies or redundancies,
- simplify the table as much as possible.

Steps for Developing Decision tables..1

- Determine the number of conditions that may affect the decision. Combine rows that overlap, for example, conditions that are mutually exclusive. The number of conditions becomes the number of rows in the top half of the decision table.

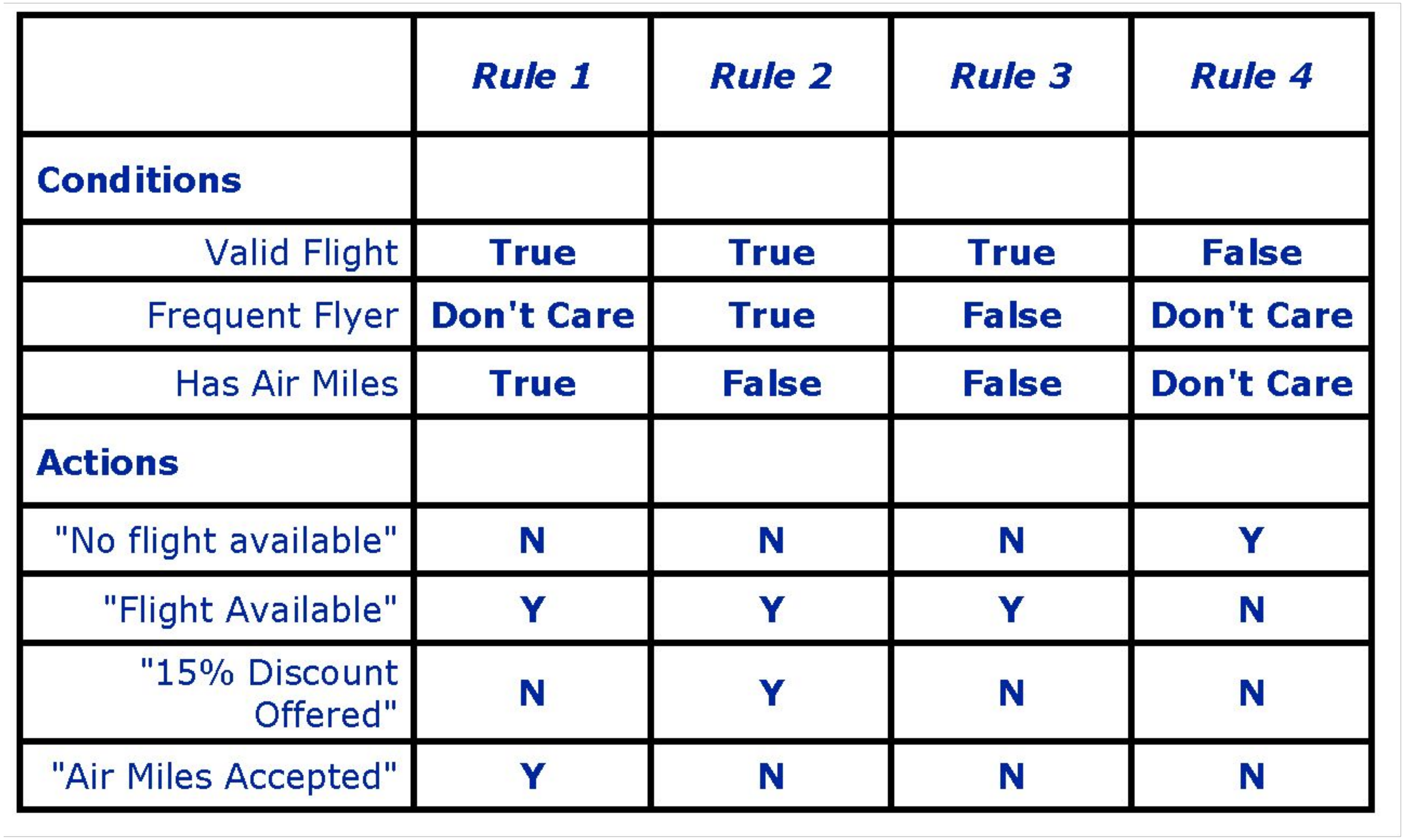
- Determine the number of possible actions that can be taken. This becomes the number of rows in the lower half of the decision table.

Steps for Developing Decision tables..2

- Determine the number of condition alternatives for each condition. In the simplest form of decision table, there would be two alternatives (Y or N) for each condition. In an extended-entry table, there may be many alternatives for each condition.
- Calculate the maximum number of columns in the decision table by multiplying the number of alternatives for each condition. If there were 4 conditions and 2 alternatives (Y or N) for each of the conditions, there would be 16 possibilities.

In a bivariate decision table, conditions are binary, restricting condition evaluations to "yes" and "no". This results in a number of columns equal to $\mathbf{2}^{\textbf{(number of conditions)}}$

Decision Table



	Rule 1	*Rule 2*	*Rule 3*	*Rule 4*
Conditions				
Valid Flight	**True**	**True**	**True**	**False**
Frequent Flyer	**Don't Care**	**True**	**False**	**Don't Care**
Has Air Miles	**True**	**False**	**False**	**Don't Care**
Actions				
"No flight available"	**N**	**N**	**N**	**Y**
"Flight Available"	**Y**	**Y**	**Y**	**N**
"15% Discount Offered"	**N**	**Y**	**N**	**N**
"Air Miles Accepted"	**Y**	**N**	**N**	**N**

Exercise

Exercises: Decision Table

Predict the expected actions for Tom, Dick and Harry

Scenario	**Expected Actions**
Tom is a frequent flyer, has air miles and selects a valid flight	
Dick selects an invalid flight	
Harry has air miles and selects a valid flight	

Testing, Quality, and Integration

Introduction

- In this section, we discuss how quality can be maintained and even enhanced using Agile tools.
- Sprints are short and do not offer extensive time for testing, therefore incorporating testing and various testing tools is of critical importance.
- One of the key tenets of Agile is “frequent verification and validation” of working software, so this section discusses different testing approaches, such as test-driven development, acceptance test-driven development, integrated testing, regression testing, and unit testing.

Introduction

- We explain the relationship test-driven development and refactoring.
- We compare manual, automated, and customer testing techniques.

Quality

- The Agile Manifesto principal “Continuous attention to technical excellence and good design enhances agility” emphasizes that quality is a central theme for Agile development.

- Some important insight behind the philosophy of creating quality code:
  - Testing does enhance quality, but it is really more of a discipline for helping developers work safely on code in “brain-sized” chunks.
  - It’s more about problem decomposition and managing cognitive load than about quality.

Quality

- On the other hand, the tests abide after the programming is done, and all of those unit tests we write while doing TDD are there to help us know if the code we are currently working on breaks any existing code.
- Likewise, the acceptance test-driven development (ATDD) tests let us know if we’ve broken any behaviour that the product’s sponsors or customers are counting on.
- Since the difficulty and cost of fixing a mistake increase the longer the defect is undetected, this is primarily a cost-saving (and face-saving) measure.
- And yet, it does increase quality.

Creating a Quality-Focused Culture

- Creating quality software is more than just using a set of tools.
- Quality starts with creating an environment where team members can do their best work.
- Creating a culture that is relentlessly focused on quality requires a lot more than just writing a quality plan and tracking to that plan, as has been the norm with organizations using a Waterfall approach or plan-driven approach.
- Quality needs to be at the basis of the organization’s culture.

Creating a Quality-Focused Culture

- The most fundamental difference between Waterfall and Agile methodologies is that Waterfall takes a more reactionary approach, whereas Agile emphasizes that teams must be proactive about quality.
- For example, in many Waterfall projects, code from the various supporting areas such as the user interface and the backend database may not be integrated until the development work is completed.
- As a result, each individual area may have done unit testing on their specific code, but they have no sense of how well that code will work with the other code that is needed to make a working product.

Creating a Quality-Focused Culture

- By the time it finally comes together in Waterfall, the code is no longer fresh in the minds of the developers and can often take longer to fix.

- The team of developers might also be suddenly overwhelmed with a huge backlog of defects that need to be fixed.

- Agile, on the other hand, advocates for daily builds that integrate all of the current code for the product.

- If the build breaks, the team is immediately alerted (often by a flashing light or other visual cue) and the developers can resolve the problem quickly, because the details of the code are fresh in their minds and there is a lot less new code to work through.

Creating a Quality-Focused Culture

- This keeps defects backlogs at a much more manageable level for the team.

- Agile teams also use specific tools such as pair programming, test-driven development, and refactoring to continually manage product quality.

Creating a Quality-Focused Culture – Pair Programming

- Pair programming offers several important benefits that contribute to the quality of the product.

- The first is that both programmers are learning from each other and becoming more familiar with different aspects of the product. This is important for continual improvement and cross-training, but also helps expedite code defects as they arise during the build.

- The second benefit is that pair programming allows each developer to focus on his or her specific role; there is a lot less cognitive overload when a developer can concentrate on getting the code working while letting the pairing partner focus on things such as performance optimization or possible bugs.

- Finally, pairing can also build trust and encourage more regular communication between team members.

Creating a Quality-Focused Culture – Test Driven Development (TDD)

- TDD requires that developers first write automated test cases and then write only the code necessary to make the test cases pass with no issues.
- This approach
  - encourages the developers to think through the requirements before writing the code,
  - encourages only the code that is needed is written,
  - and ensures that each piece of the code has gone through an initial quality check before formal testing.

Creating a Quality-Focused Culture – Test Driven Development (TDD)

- These test cases are not used only for unit testing code during development, but also to create an automated test suite used to regularly test the product.

Creating a Quality-Focused Culture - Refactoring

- Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure.
- Refactoring does not mean you are debugging your code, but rather cleaning it up so that extraneous code is removed and complies with other patterns or structures that may exist in the product.
- Refactoring is a process to ensure that the least amount of code is written to pass a failing test.

Creating a Quality-Focused Culture - Refactoring

- The idea of refactoring originated from the mathematical term equivalence, where equations are continually reduced until no additional reductions can be made.

- The same is true with code: You are not ultimately changing the functions, but you are reducing them to the simplest form possible that keeps the code functioning as it was intended.

Creating a Quality-Focused Culture - Refactoring

- The idea of coupling comes into play when you are refactoring code.
- Coupling is the extent to which a program module depends on other modules within the program.
- It is encouraged to decouple the code, or make each piece less dependent on other pieces and more individually testable.
- The primary benefit of decoupling is that it reduces the number of changes that are required if you want to use a different module.

Minimizing Defect Backlog

- The best-case scenario when creating a product is to carefully write the code so that defects are never introduced.

- Although most developers would prefer to spend all of their time writing new and innovative code, to err is human, and defects happen in even the most thoughtfully developed products.

- To create a quality product, there has to be a continuous focus on defect detection during all phases of development, from finding a bug while pairing to running regular integration tests.

Minimizing Defect Backlog

- Some companies recommend that fixing defects always takes priority over writing new code.
- It is argued that the longer you wait to fix a bug, the more the company will ultimately pay in time and money.
- Developers can usually find and fix bugs much faster if the code is fresh in their minds.
- If it has been months since a developer looked at a section of the code, then it is highly likely that another team member has modified that code, making it more difficult to fix.
- If a bug is found in the field, you risk a hit to product reputation and the loss of valuable customers.

Minimizing Defect Backlog

- Defects are typically found in the early phases of development during unit testing, where developers test their own code, through paired programming, or by using TDD.
- These are important points to find defects, but these bugs are found in isolated pieces of the code.
- Often many of the defects arise when the code from various parts of the product come together as an integrated product; this is known as **integration build**.

Minimizing Defect Backlog

- Integration builds can happen whenever code is completed.
- However, many Waterfall projects have traditionally used weekly (or monthly) scheduled builds.
- The thought behind weekly builds was that they would collect all of the code that was changed or added for that week and come in Monday to review any bugs that may have been introduced.
- Some of the Waterfall teams used even less regular build schedules and waited until all of the code was completed before they did an integration build.
- You can imagine the defect backlog they were faced with when a month’s worth of development work finally came together.

Minimizing Defect Backlog

- Consistent with the Agile Manifesto, which states, “Working software is the primary measure of progress,” Agile promotes continual integration builds.

- This means that as soon as the code is checked in, it is integrated with the overall product build.

- This does not mean the updates are part of the live product; it simply means that a new integrated build is always available to the product development team for testing.

Minimizing Defect Backlog

- Some Agile teams are so focused on finding defects quickly that they have implemented a system where a siren or alarm goes off when new code breaks a build.

- The siren alerts the entire team that there is a problem and allows them to respond immediately.

- Developers do not want their code to be the one that causes the embarrassing noise, so they are motivated to carefully test before they check anything in for a build.

Manual, Automated, and Customer Testing

- A product development team has two options when testing whether the code is working as it was designed.

- The first is **manual testing** and the other option is **automated testing**.

Manual, Automated, and Customer Testing

- Manual testing is where a human tester must progress through each step or look over the product to make sure nothing about the code or design is defective.

- Manual testing can be very time-consuming and is subject to human error, but is often necessary in cases where the feature cannot be automated or if it is a visual update such as a background colour that must match the rest of the product.

- Manual testing is often used when validation of the user interface is required.

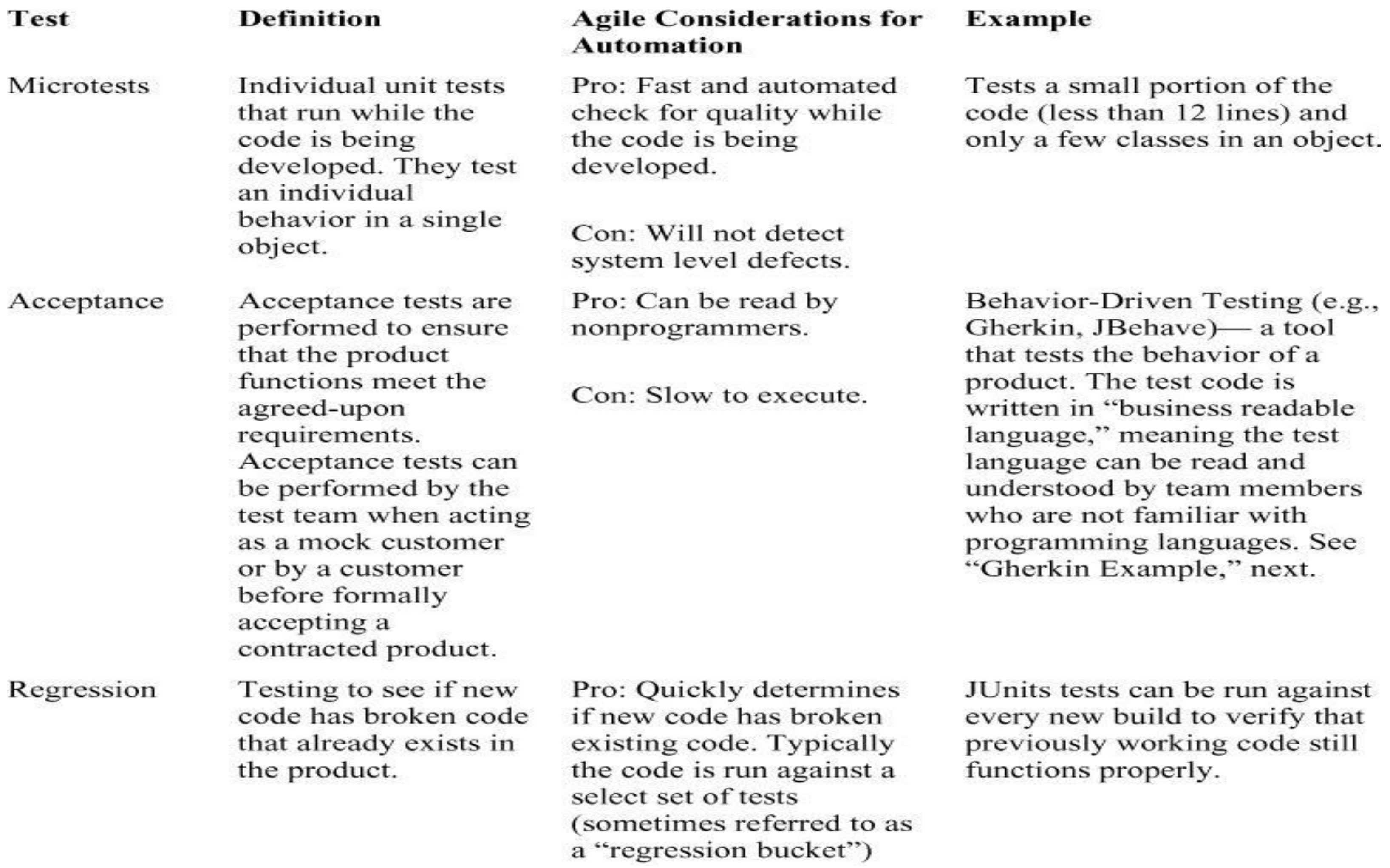
Manual, Automated, and Customer Testing

- Automated testing uses software that is independent of the software being tested to execute tests without human intervention and compare the results against the desired outcome.

- A single test or a string of tests can be automated, but running a series of tests tends to provide the most benefit.

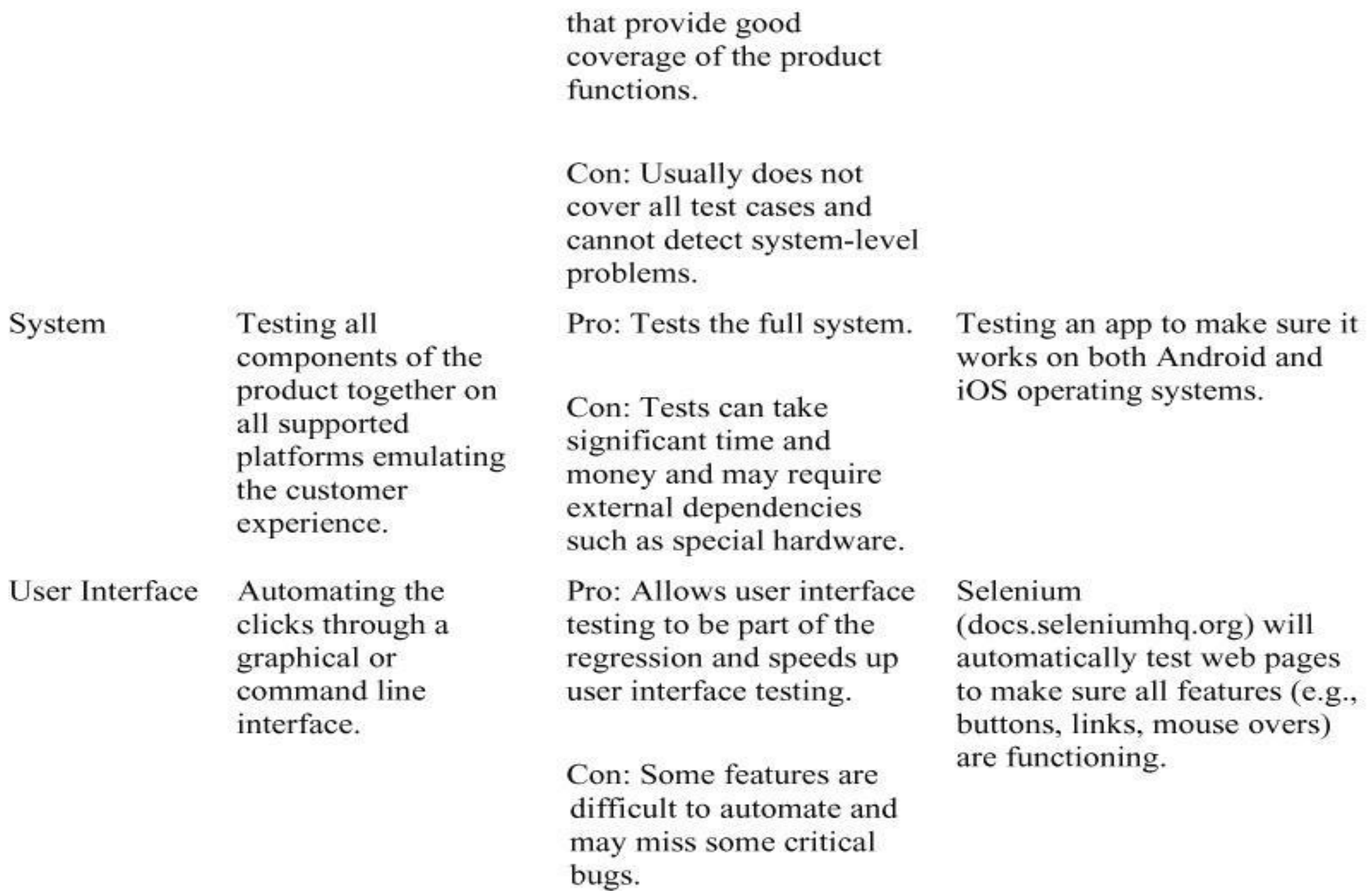
- An automated test can be anything such as a series of clicks in a user interface, execute commands in a command line interface, or checks that data is appropriately stored in a database.

Types of Testing That Benefit from Automation



Test	Definition	Agile Considerations for Automation	Example
Microtests	Individual unit tests that run while the code is being developed. They test an individual behavior in a single object.	Pro: Fast and automated check for quality while the code is being developed. Con: Will not detect system level defects.	Tests a small portion of the code (less than 12 lines) and only a few classes in an object.
Acceptance	Acceptance tests are performed to ensure that the product functions meet the agreed-upon requirements. Acceptance tests can be performed by the test team when acting as a mock customer or by a customer before formally accepting a contracted product.	Pro: Can be read by nonprogrammers. Con: Slow to execute.	Behavior-Driven Testing (e.g., Gherkin, JBehave)— a tool that tests the behavior of a product. The test code is written in "business readable language," meaning the test language can be read and understood by team members who are not familiar with programming languages. See "Gherkin Example," next.
Regression	Testing to see if new code has broken code that already exists in the product.	Pro: Quickly determines if new code has broken existing code. Typically the code is run against a select set of tests (sometimes referred to as a "regression bucket")	JUnits tests can be run against every new build to verify that previously working code still functions properly.

Types of Testing That Benefit from Automation



		that provide good coverage of the product functions. Con: Usually does not cover all test cases and cannot detect system-level problems.	
System	Testing all components of the product together on all supported platforms emulating the customer experience.	Pro: Tests the full system. Con: Tests can take significant time and money and may require external dependencies such as special hardware.	Testing an app to make sure it works on both Android and iOS operating systems.
User Interface	Automating the clicks through a graphical or command line interface.	Pro: Allows user interface testing to be part of the regression and speeds up user interface testing. Con: Some features are difficult to automate and may miss some critical bugs.	Selenium (docs.seleniumhq.org) will automatically test web pages to make sure all features (e.g., buttons, links, mouse overs) are functioning.

Gherkin/Cucumber

- Gherkin is the language that Cucumber understands.

- It is a Business Readable, Domain Specific Language that lets you describe software’s behaviour without detailing how that behaviour is implemented.

Gherkin/Cucumber

- To write the test in Gherkin language, you first need to describe the feature in the “Feature” line and the scenario they are testing in the “Scenario” line.

- From there you use the “Given” line to describe the current condition and the “And,” “When,” and “Then” lines to explain how the scenario proceeds from there.

Gherkin/Cucumber Example

Feature: The game returns a null winner when both the dealer and the player have blackjack.

Scenario: Both the dealer and the player have blackjack

Given the initial round of cards have been dealt

And neither player has busted

When the dealer and player review their cards

And the dealer has blackjack

And the player has blackjack

Then the game returns a null winner

And neither the dealer nor the player wins the game

Gherkin/Cucumber Example

- If the product owner and the business analyst agree that the code accurately represents their goals from the user story, the developers proceed to include these tests in their automated system tests to ensure the product continues to operate as designed.

- Refer to https://cucumber.io/ for details of the steps.

Customer Feedback

- Customer feedback can take many forms.
- Alpha tests are used to allow customers to try early versions of the code.
- Beta tests are similar in that they allow customers to try out the product before it is released generally to the public, but the product has been through much more extensive testing by the development team.
- Usability testing involves observing the customer interacting with the product to understand where there may be opportunities to improve the ease of use.

Customer Feedback

- Alpha, beta, and usability testing are not unique to Agile and have been used for years during software development projects.
- Agile does advocate for new techniques that involve regular engagement with the customer.
- In some of the more extreme cases, the customers are actually part of the development team.

Customer Feedback

- Some companies use what is often called a *customer council*, where the development team meets with the customers to review ideas or working code.
- Other companies invite the customers to participate in the stakeholder feedback sessions.
- No matter what approach you use to get customer feedback, it is important to remember the Agile Manifesto value that emphasizes, “It is more important that the customer become intimately involved with the product development team than to focus on the terms and conditions of the project.”

Conclusion

- One of the most important benefits of moving to a more Agile approach to developing software is the emphasis on quality.
- Tools such as pair programming, test-driven development, and refactoring offer techniques that build in quality from the beginning.
- Introducing automated testing and regular customer feedback are also best practices that help ensure issues are found early in the process and can be addressed quickly.

Clean Code

What Is Clean Code?

- Code should be elegant and efficient,
- The logic should be straightforward to make it hard for bugs to hide,
- The dependencies minimal to ease maintenance, and

- Error handling complete according to an articulated strategy.

Bjarne Stroustrup, inventor of C++ and author of *The C++ Programming Language*

What Is Clean Code?

- Clean code is simple and direct.
- Clean code reads like well-written prose.
- Clean code never obscures the designer’s intent but rather is full of crisp abstractions and straightforward lines of control.
- Grady makes some of the same points as Bjarne, but he takes a *readability* perspective.

Grady Booch, author of *Object Oriented Analysis and Design with Applications*

What Is Clean Code?

- Clean code can be read, and enhanced by a developer other than its original author.
- It has unit and acceptance tests.
- It has meaningful names.
- It provides one way rather than many ways for doing one thing.
- Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone.
- Big Dave shares Grady's desire for readability, but with an important twist. Dave asserts that clean code makes it easy for *other* people to enhance it.

"Big" Dave Thomas, founder of OTI, godfather of the Eclipse strategy

What Is Clean Code?

- Clean code always looks like it was written by someone who cares. There is nothing obvious that you can do to make it better.
- For Michael the key word is **care**. Clean code is code that has been taken care of. Someone has taken the time to keep it simple and orderly. They have paid appropriate attention to details. They have cared.

Michael Feathers, author of *Working Effectively with Legacy Code*

What Is Clean Code?

- Ron endorses Beck's (http://c2.com/cgi/wiki?KentBeck) rules of simple code. In priority order, simple code:
  - Runs all the tests;
  - Contains no duplication;
  - Expresses all the design ideas that are in the system;
  - Minimizes the number of entities such as classes, methods, functions, and the like.

Ron Jeffries, author of *Extreme Programming Installed* and *Extreme Programming Adventures in C#*

The Boy Scout Rule

- It’s not enough to write the code well. The code has to be *kept clean* over time. If care is not taken code rots and degrades as time passes. So we must take an active role in preventing this degradation.
- The Boy Scouts of America have a simple rule that we can apply to our profession: Leave the campground cleaner than you found it.

- If we all checked-in our code a little cleaner than when we checked it out, the code simply could not rot.

- The cleanup doesn’t have to be something big. Change one variable name for the better, break up one function that’s a little too large, eliminate one small bit of duplication, clean up one composite if statement.

Meaningful Names

Introduction

- Names are everywhere in software.
- We name our variables, our functions, our arguments, classes, and packages.
- We name our source files and the directories that contain them.
- We name our jar files and war files and ear files.
- Because we do so much of it, we’d better do it well.
- What follows are some simple rules for creating good names.

1. Use Intention-Revealing Names

- Choosing good names takes time but saves more than it takes.

- So take care with your names and change them when you find better ones.

- Everyone who reads your code (including you) will be happier if you do.

1. Use Intention-Revealing Names

- The name of a variable, function, or class, should answer all the big questions.
- It should tell you why it exists, what it does, and how it is used.
- If a name requires a comment, then the name does not reveal its intent.
  - e.g. *int d; // elapsed time in days*

1. Use Intention-Revealing Names

- We should choose a name that specifies what is being measured and the unit of that measurement:
  - e.g. *int elapsedTimeInDays;*
  - e.g. *int daysSinceCreation;*

- Choosing names that reveal intent can make it much easier to understand and change code.

2. Avoid Disinformation

- Beware of using names which vary in small ways.

- How long does it take to spot the subtle difference between a *XYZControllerForEfficientHandlingOfStrings* in one module and, in another module, *XYZControllerForEfficientStorageOfStrings*?

- The words have frightfully similar shapes.

2. Avoid Disinformation

- This can be a problem with modern Java environments where we enjoy automatic code completion.
- We write a few characters of a name and press some hotkey combination (if that) and are rewarded with a list of possible completions for that name.
- It is very helpful if names for very similar things sort together alphabetically and if the differences are very obvious, because the developer is likely to pick an object by name without seeing your copious comments or even the list of methods supplied by that class.

3. Use Pronounceable Names

- Humans are good at words.
- A significant part of our brains is dedicated to the concept of words.
- And words are, by definition, pronounceable.
- It would be a shame not to take advantage of that huge portion of our brains that has evolved to deal with spoken language. So make your names pronounceable.

3. Use Pronounceable Names

Change Unpronounceable Names to ---->

Pronounceable Names

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint =
"102";
  /* ... */
};
```

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;;
  private final String recordId = "102";
  /* ... */
};
```

4. Use Searchable Names

- Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text.
- One might easily search for MAX_CLASSES_PER_STUDENT, but the number 7 could be more troublesome.

- Searches may turn up the digit as part of file names, other constant definitions, and in various expressions where the value is used with different intent.

- It is even worse when a constant is a long number and someone might have transposed digits, thereby creating a bug while simultaneously evading the programmer's search.

4. Use Searchable Names

- Likewise, the name e is a poor choice for any variable for which a programmer might need to search.
- It is the most common letter in the English language and likely to show up in every passage of text in every program.
- In this regard, longer names trump shorter names, and any searchable name trumps a constant in code.
- My personal preference is that single-letter names should not be used although others advocate using them as local variables inside short methods.

4. Use Searchable Names

Compare This

```
for (int j=0; j<34; j++) {
  s += (t[j]*4)/5;
}
```

To This

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] *
realDaysPerIdealDay;
  int realTaskWeeks = (realTaskdays /
WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

5. Class Names

- Classes and objects should have noun or noun phrase names like Customer, WikiPage, Account, and AddressParser.

- A class name should not be a verb.

6. Method Names

- Methods should have verb or verb phrase names like *postPayment*, *deletePage*, or *save*.

- Accessors, mutators, and predicates should be named for their value and prefixed with get, set e.g.
  - *string name = employee.getName();*
  - *customer.setName("mike");*

Summary

- The hardest thing about choosing good names is that it requires good descriptive skills and a shared cultural background.

- This is a teaching issue rather than a technical, business, or management issue.

- As a result many people in this field don’t learn to do it very well.

Functions (Methods)

Introduction

- Functions or methods are the first line of organization in any program.
- Writing them well is the focus of this section.

1. Small

- The first rule of functions or methods is that they should be small.
- The second rule of functions is that *they should be smaller than that*.
- This is not a statement that I can justify.
- Experience has taught me, through long trial and error, is that functions should be very small.

1. Small

- How short should a function be?

- Each function should be transparently obvious.

- Each function should tell a story.
- And each led you to the next in a compelling order.
- *That's* how short your functions should be

2. Blocks and Indenting

- This implies that the blocks within if statements, else statements, while statements, and so on should be one line long.
- Probably that line should be a function call.
- Not only does this keep the enclosing function small, but it also adds documentary value because the function called within the block can have a nicely descriptive name.

2. Blocks and Indenting

- This also implies that functions should not be large enough to hold nested structures.
- Therefore, the indent level of a function should not be greater than one or two.
- This, of course, makes the functions easier to read and understand

3. Do One Thing

- Functions should do one thing.
- They should do it well.
- They should do it only.
- A way to know that a function is doing more than “one thing” is if you can extract another function from it.

4. Switch Statements

- It’s hard to make a small switch statement.
- Even a switch statement with only two cases is larger than I’d like a single block or function to be.
- It’s also hard to make a switch statement that does one thing.
- By their nature, switch statements always do *N* things.

4. Switch Statements

- Unfortunately we can’t always avoid switch statements, but we *can* make sure that each switch statement is buried in a low-level class and is never repeated.
- We do this, of course, with polymorphism.

(Note: Polymorphism)

- Generally, the ability to appear in many forms.
- In object-oriented programming, *polymorphism* refers to a programming language's ability to process objects differently depending on their data type or class. More specifically, it is the ability to redefine *methods* for *derived classes.*
- For example, given a base class *shape,* polymorphism enables the programmer to define different *area* methods for any number of derived classes, such as circles, rectangles and triangles. No matter what shape an object is, applying the *area* method to it will return the correct results.

4. Switch Statements

- Consider the following piece of code:

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
    switch (e.type) {
      case COMMISSIONED:
        return calculateCommissionedPay(e);
      case HOURLY:
        return calculateHourlyPay(e);
      case SALARIED:
        return calculateSalariedPay(e);
      default:
        throw new InvalidEmployeeType(e.type);
    }
}
```

4. Switch Statements

- There are several problems with this function.
- First, it’s large, and when new employee types are added, it will grow.
- Second, it very clearly does more than one thing.
- Third, it violates the Single Responsibility Principle (SRP) because there is more than one reason for it to change.
- Fourth, it violates the Open Closed Principle (OCP) because it must change whenever new types are added.

4. Switch Statements

- Finally other functions that will have the same structure.

- For example we could have

  isPayday(Employee e, Date date),

  or

  deliverPay(Employee e, Money pay).

4. Switch Statements

- The solution to this problem (see next slide) is to bury the switch statement in the basement of an ABSTRACT FACTORY and never let anyone see it.

- The factory will use the switch statement to create appropriate instances of the derivatives of Employee, and the various functions, such as calculatePay, isPayday, and deliverPay, will be dispatched polymorphically through the Employee interface.

4. Switch Statements

```
public abstract class Employee {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
```

```
public interface EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r)
throws InvalidEmployeeType;
}
```

```
public class EmployeeFactoryImpl
implements EmployeeFactory {

public Employee
makeEmployee(EmployeeRecord r) throws
InvalidEmployeeType {
    switch (r.type) {
      case COMMISSIONED:
        return new CommissionedEmployee(r) ;
      case HOURLY:
        return new HourlyEmployee(r);
      case SALARIED:
        return new SalariedEmploye(r);
      default:
        throw new InvalidEmployeeType(r.type);
    }
  }
}
```

4. Switch Statements

- My general rule for switch statements is that they can be tolerated if they appear only once, are used to create polymorphic objects, and are hidden behind an inheritance relationship so that the rest of the system can’t see them.

- Of course every circumstance is unique, and there are times when I violate one or more parts of that rule.

5. Use Descriptive Names

- It is hard to overestimate the value of good names.
- Remember : “*You know you are working on clean code when each routine turns out to be pretty much what you expected.*”

- Half the battle to achieving that principle is choosing good names for small functions that do one thing.

- The smaller and more focused a function is, the easier it is to choose a descriptive name.

5. Use Descriptive Names

- Don’t be afraid to make a name long.
- A long descriptive name is better than a long descriptive comment.
- Use a naming convention that allows multiple words to be easily read in the function names, and then make use of those multiple words to give the function a name that says what it does.
- Don’t be afraid to spend time choosing a name.

5. Use Descriptive Names

- Choosing descriptive names will clarify the design of the module in your mind and help you to improve it.

- It is not at all uncommon that hunting for a good name results in a favorable restructuring of the code.

6. Function Arguments

- The ideal number of arguments for a function is zero (niladic).
- Next comes one (monadic), followed closely by two (dyadic).
- Three arguments (triadic) should be avoided where possible.
- More than three (polyadic) requires very special justification—and then shouldn’t be used anyway.

6. Function Arguments

- Arguments are hard.
- They take a lot of conceptual power.
- Arguments are even harder from a testing point of view.
- Imagine the difficulty of writing all the test cases to ensure that all the various combinations of arguments work properly.

6. Function Arguments

- If there are no arguments, this is trivial.
- If there’s one argument, it’s not too hard.
- With two arguments the problem gets a bit more challenging.
- With more than two arguments, testing every combination of appropriate values can be daunting.

6. Function Arguments

- Output arguments (pass by reference) are harder to understand than input arguments.

- When we read a function, we are used to the idea of information going *in* to the function through arguments and *out* through the return value.

- We don’t usually expect information to be going out through the arguments.

- So output arguments often cause us to do a double-take.

7. Flag Arguments

- Flag arguments are ugly.
- Passing a boolean into a function is a truly terrible practice.
- It immediately complicates the signature of the method, loudly proclaiming that this function does more than one thing.
- It does one thing if the flag is true and another if the flag is false!

8. Dyadic Functions

- A function with two arguments is harder to understand than a monadic function.
- For example, *writeField(name)* is easier to understand than *writeField(output-Stream, name).*
- Though the meaning of both is clear, the first glides past the eye, easily depositing its meaning.
- The second requires a short pause until we learn to ignore the first parameter.

8. Dyadic Functions

- And *that*, of course, eventually results in problems because we should never ignore any part of code.

- The parts we ignore are where the bugs will hide.

8. Dyadic Functions

- There are times, of course, where two arguments are appropriate.
- For example, *Point p = new Point(0,0);* is perfectly reasonable.
- Cartesian points naturally take two arguments.
- However, the two arguments in this case *are ordered components of a single value!* Whereas output-Stream and name have neither a natural cohesion, nor a natural ordering.

8. Dyadic Functions

- Even obvious dyadic functions like *assertEquals(expected, actual)* are problematic.
- How many times have you put the actual where the expected should be?
- The two arguments have no natural ordering.
- The expected, actual ordering is a convention that requires practice to learn.

8. Dyadic Functions

- Dyads aren’t evil, and you will certainly have to write them.
- However, you should be aware that they come at a cost and should take advantage of what mechanisms may be available to you to convert them into monads.
- For example, you might make the *writeField* method a member of *outputStream* so that you can say *outputStream. writeField(name).*

9. Triads

- Functions that take three arguments are significantly harder to understand than dyads.
- The issues of ordering, pausing, and ignoring are more than doubled.
- I suggest you think very carefully before creating a triad.

9. Argument Objects

- When a function seems to need more than two or three arguments, it is likely that some of those arguments ought to be wrapped into a class of their own.

- Consider, for example, the difference between the two following declarations:

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

9. Argument Objects

- Reducing the number of arguments by creating objects out of them may seem like cheating, but it’s not.

- When groups of variables are passed together, the way x and y are in the example above, they are likely part of a concept that deserves a name of its own.

10. Verbs and Keywords

- Choosing good names for a function can go a long way toward explaining the intent of the function and the order and intent of the arguments.

- In the case of a monad, the function and argument should form a very nice verb/noun pair.

- For example, *write(name)* is very evocative. Whatever this “name” thing is, it is being “written.”

- An even better name might be writeField(name), which tells us that the “name” thing is a “field.”

10. Verbs and Keywords

- This last is an example of the *keyword* form of a function name.

- Using this form we encode the names of the arguments into the function name.

- For example, *assertEquals* might be better written as *assertExpectedEqualsActual(expected, actual)*.

- This strongly mitigates the problem of having to remember the ordering of the arguments.

11. Prefer Exceptions to Returning Error Codes

- When you return an error code, you create the problem that the caller must deal with the error immediately.
- On the other hand, if you use exceptions instead of returned error codes, then the error processing code can be separated from the normal path code and can be simplified.

12. Extract Try/Catch Blocks

- *Try/catch* blocks are ugly in their own right.
- They confuse the structure of the code and mix error processing with normal processing.
- So it is better to extract the bodies of the *try* and *catch* blocks out into functions of their own.

12. Extract Try/Catch Blocks

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  }
  catch (Exception e) {
    logError(e);
  }
 }

 private void deletePageAndAllReferences(Page page) throws Exception {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
 }

 private void logError(Exception e) {
  logger.log(e.getMessage());
 }
```

12. Extract Try/Catch Blocks

- In the code on the previous slide, the <u>*delete*</u> function is all about error processing.
- It is easy to understand and then ignore.
- The <u>*deletePageAndAllReferences*</u> function is all about the processes of fully deleting a page.
- Error handling can be ignored.
- This provides a nice separation that makes the code easier to understand and modify.

13. Error Handling Is One Thing

- Functions should do one thing.
- Error handing is one thing.
- Thus, a function that handles errors should do nothing else.
- This implies (as in the previous example) that if the keyword *try* exists in a function, it should be the very first word in the function and that there should be nothing after the *catch/finally* blocks.

14. Structured Programming

- Some programmers follow Edsger Dijkstra’s rules of structured programming.
- Dijkstra said that every function, and every block within a function, should have one entry and one exit.
- Following these rules means that there should only be one return statement in a function, no break or continue statements in a loop, and never, *ever*, any goto statements.

14. Structured Programming

- While we are sympathetic to the goals and disciplines of structured programming, those rules serve little benefit when functions are very small.

- It is only in larger functions that such rules provide significant benefit.
- So if you keep your functions small, then the occasional multiple return, break, or continue statement does no harm and can sometimes even be more expressive than the single-entry, single-exit rule.

- On the other hand, goto only makes sense in large functions, so it should be avoided.

Continuous Integration and Deployment

Integrate at least daily

- Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early.
- By integrating regularly, you can detect errors quickly, and locate them more easily.

Solve problems quickly

- Because you’re integrating so frequently, there is significantly less back-tracking to discover where things went wrong, so you can spend more time building features.
- Continuous Integration is cheap. Not continuously integrating is costly. If you don’t follow a continuous approach, you’ll have longer periods between integrations. This makes it exponentially more difficult to find and fix problems. Such integration problems can easily knock a project off-schedule, or cause it to fail altogether.

Continuous Integration brings multiple benefits

- Say goodbye to long and tense integrations
- Increase visibility which enables greater communication
- Catch issues fast and nip them in the bud
- Spend less time debugging and more time adding features
- Proceed in the confidence you’re building on a solid foundation
- Stop waiting to find out if your code’s going to work
- Reduce integration problems allowing you to deliver software more rapidly

More than a process - *The Practices*

- Maintain a single source repository
- Automate the build
- Make your build self-testing
- Every commit should build on an integration machine
- Keep the build fast
- Test in a clone of the production environment
- Make it easy for anyone to get the latest executable
- Everyone can see what’s happening
- Automate deployment

More than a process - *How to do it*

- Developers check out code into their private workspaces.
- When done, commit the changes to the repository.
- The CI server monitors the repository and checks out changes when they occur.
- The CI server builds the system and runs unit and integration tests.
- The CI server releases deployable artefacts for testing.
- The CI server assigns a build label to the version of the code it just built.
- The CI server informs the team of the successful build.
- If the build or tests fail, the CI server alerts the team.
- The team fix the issue at the earliest opportunity.
- Continue to continually integrate and test throughout the project.

More than a process - *Team Responsibilities*

- Check in frequently
- Don’t check in broken code
- Don’t check in untested code
- Don’t check in when the build is broken
- Don’t go home after checking in until the system builds
- Many teams develop rituals around these policies, meaning the teams effectively manage themselves, removing the need to enforce policies from on high.

More than a process - *Continuous Deployment*

- Continuous Deployment is closely related to Continuous Integration and refers to the release into production of software that passes the automated tests.

- By adopting both Continuous Integration and Continuous Deployment, you not only reduce risks and catch bugs quickly, but also move rapidly to working software.