People-Centered Software Development: An Overview of Agile Methodologies

Frank Maurer and Theodore D. Hellmann

The University of Calgary, Department of Computer Science,
2500 University Drive NW, Calgary, Alberta, Canada
{frank.maurer, tdhellma}@ucalgary.ca

Abstract. This chapter gives an overview of agile software development processes and techniques. The first part of the chapter covers the major agile project management techniques with a focus on project planning. Iteration planning and interaction design approaches are given special focus. The second part of the chapter covers agile quality assurance with a focus on test-driven development and the state space of testing. Current problems in agile testing, including measuring test quality and testing applications with large state spaces, are discussed.

Keywords: Agile Methods, Agile Project Management, Agile Interaction Design, Test-Driven Development, State Space Testing

1 Introduction

Software development is a complex undertaking that poses substantial challenges to teams in industry. In the 1980ies, companies tried to use Computer-Aided Software Engineering (CASE) tools to increase the efficiency of software development processes. The core idea was to use graphical notations to describe the functionality of a software system on an abstract level and then generate (most of) the code from it. However, the success of these approaches was limited and software teams nowadays still write code manually.

In the 1990ies, software developers were a scarce resource and companies focused on improving the development process to optimize their development efforts. Software process improvement (SPI) initiatives following CMMI, ISO 900x or SPICE ideas were commonplace. SPI approaches basically require an organization to define the steps and outcomes of each development step and then ensure that all teams are following these best practices: document what you do and do what is documented. As a side effect of the process definition, organizations often adopted Tayloristic[1] waterfall processes where steps in the process corresponded to roles in the organization and

[1] In his seminal 1911 book "The Principles of Scientific Management", Frederick Taylor discussed repeatable manufacturing processes with a strong division of labor and a separation between manufacturing and engineering work.

handoffs between steps happened in the form of documents. Unfortunately, many SPI implementations resulted in heavyweight, document-centric processes that created a substantial overhead for software development teams.

Agile processes are trying to swing the pendulum back. Proponents of agile methods ask the questions: how can we refocus projects on the bare minimum required to make software development effective and efficient? What does a software development team really have to do to create business value?

Agile methods came to the forefront of the discussion in the software development community in the late 1990ies and have been widely adopted since then. Initially, in the late 1990ies and early 2000s, teams started their journey by using ideas from extreme programming (XP) to improve their engineering processes. Test-driven development, pair programing, continuous integration, short release cycles, refactoring, simple design and on-site customer are techniques that were included in Kent Beck's XP book [1]. XP introductions often happened bottom-up: software developers pushed the ideas into their development projects and hoped to streamline the delivery of value to their customers.

By the mid 2000s, agile methods moved from the development cubicle to the front-line management level. At that time, many teams started their agile adoption with ideas from Scrum for improving the management of software projects. Ken Schwaber's Scrum [2] emphasizes iterative and incremental development, self-organizing teams and continuous process improvement in small steps. The methodology provides a set of tools to help with coordinating software development efforts while ensuring that value is delivered to customers frequently and reliably. This focus on project management issues made Scrum a favorite for front-line and middle management – which resulted in a middle-out strategy for agile method adoption where middle managers pushed agile ideas downwards into their teams as well as upwards into senior management.

More recently, in the late 2000s/early 2010s, agile adoptions often seem to be pushed from senior management to the whole enterprise. Mary & Tom Poppendieck's Lean Software Development [3] is based on ideas from the Toyota Production System and translates them into software development processes. Lean software development provides guidelines for enterprise-level agile adoptions and includes techniques like value stream mapping, flow, reducing cycle time and kanban.

While we highlighted XP, Scrum and Lean above, other methodologies fall into the agile space and had substantial impact on the area. Feature-driven development [4], DSDM [5], Crystal Clear [6], and adaptive software development [7] are some of the approaches that had a substantial impact on the thinking and progress in the agile community. However, our own – subjective – observations with industrial partners clearly indicate that XP, Scrum and Lean are the ones that are more widely adopted and discussed.

When we interact with teams that want to adopt agile approaches, we usually suggest they initially focus on two aspects: agile project management and agile quality assurance. The remainder of this chapter discusses these in more detail.

In Section 2, we provide an overview of agile project management approaches. We discuss user stories, user story mapping, and low-fidelity prototyping as well as re-

lease and iteration planning. Section 3 presents an overview of agile quality assurance focusing on test-driven development and acceptance test-driven development, and also discusses the increasingly-important topic of graphical user interface (GUI) testing. The concept of the state space of an application as it relates to testing is also described in Section 3, as well as the implications of this concept in relation to GUI testing. The final section summarizes our findings.

2 Agile Project Management

Agile project management is based on four values:

- Communication,
- Simplicity,
- Feedback, and
- Courage.

Communication is key for any software development project. Business representatives understand their problems and can develop ideas about how they can be overcome with software. However, they usually do not have the technical skills to develop the software system. Thus, communication is an essential bridge between the business domain and the development domain. Communication is needed between all stakeholders in a project – from senior management to future users, IT operations, software development, user experience, project management.

Simplicity is about asking the question: what is the simplest thing that could possibly work? The question needs to be raised when designing software to avoid gold-plating and over-engineering – YAGNI (you ain't gonna need it) is the agile battle cry. But it also needs to be raised in regard to project planning and progress tracking: what does a team have to do to get an accurate picture of the future development effort?

Feedback is fast and frequent in agile teams. Essential feedback comes from putting the system (or updates) into production as quickly as possible. Feedback from real use allows the development team to find bugs early and fix them. It helps the team to steer the project back onto the right path when needed and provides necessary confirmation of success when users *do not* find problems with newly deployed features. Feedback from successful regression testing provides validation that existing features have not been broken by new development results – ensuring the effort estimates remain valid and the project stays on track.

Courage is needed when developers point out unrealistic expectations to customers: not everything can be delivered by a few weeks of work. Courage is also essential when the development team has to explain to the customer why delivering new features must be postponed for a major redesign of the existing platform.

A core agile strategy that embodies the four agile values is the creation of holistic teams.

2.1 Whole Team

A primary goal in agile project management is to create a "whole team" that has all skills required to successfully create a software system. The team usually includes business stakeholders, analysts, software architects and designers, developers, testers, as well as any other stakeholder that needs to be involved in the discussions. Some agile methods, for example XP, argue that teams usually require multi-skilled personnel: generalists that can fulfill multiple roles for the team. In such teams, role rotation is common. However, teams –specifically larger teams – often include specialists that focus on certain aspects of the project. Depending on workload, specialists are shared between multiple teams, e.g. database administrators or usability experts often serve in their respective roles in multiple teams. The whole team is involved in collaboratively planning the next steps in the development effort. If possible, project planning is conducted by bringing all team members into the same room for a face-to-face conversation.

2.2 Project Management

Project management deals with four variables: cost, scope, schedule and quality.

Cost in software development is highly correlated to the number and quality of team members. Cost overruns were – and still are – a major problem for software development projects. The scope of a project is defined by the set of all features that need to be delivered to the customer. The schedule determines when a feature is or should be delivered. The customer perception of quality is based on fitness for purpose as well as the number of bugs that are found after delivery. Project management needs to determine the appropriate balance between these dimensions. Improvements in one dimension often impact other dimensions; for example, reducing the time to delivery can to a certain extent be accomplished by hiring additional developers for the duration of the project – which makes the project more expensive. It is a fallacy that project management can optimize each dimension individually.

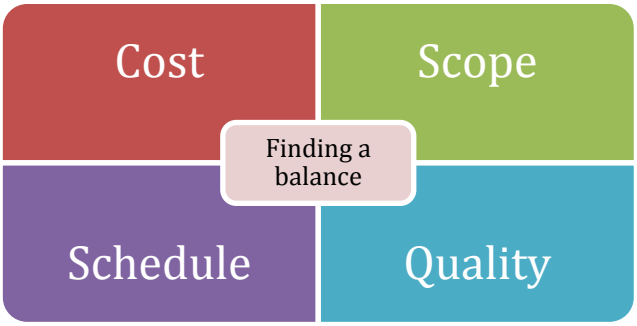

Fig. 1. Project variables

Agile methods recommend against spending much effort on upfront work. After acquiring a basic understanding of the project's goals and high-level requirements, teams are expected to quickly start development iterations that deliver potentially shippable product functionality. Usually, upfront work is limited to days or a few weeks of effort. This approach is quite the opposite of more traditional software development processes that front-load the development process and emphasize a thorough and detailed analysis of software requirements followed by substantial architectural and design work. The benefits of the agile approach are:

- As business environments and processes change quickly in today's competitive environment, a large delay between determining a requirement and delivering it might make this requirement obsolete. In this sense, the agile approach minimizes the risk that effort is spent on analyzing, designing and implementing features that will be unnecessary by the time they are delivered.
- The limited amount of development effort available in short iterations naturally forces business stakeholders to prioritize their feature requests. Reasonable businesspeople understand that a team will not be able to deliver all their requirements in the next few weeks and will determine what features are most urgently needed. As a result, requirements tend to be fulfilled in decreasing levels of importance or urgency. This in turn allows management to cut off a project when it determines that the business value of future iterations does not justify the costs incurred by them.
- As requirements are quickly turned into implemented features, feedback from actual use helps to determine if these features are what is actually needed or need to be revised.
- Effort spent on upfront work is actually wasted if the system is never delivered to production. Limiting work before delivering a first feature set to production reduces this risk.
- Source code is where the rubber hits the road in software development. Detailed analysis and design models that are unrealistic exist – but the first attempt to build the system often finds their issues quickly.

However, proponents of more upfront-centric approaches have arguments that can be seen as a criticism of the agile style:

- Empirical studies have shown that fixing a bug after the software is delivered is 60-100 times more expensive than fixing the same issue in the analysis phase [8]. Thus, a thorough process that emphasizes analysis and design will save expenses, as it does not allow bugs to slip through.
- Assuming the software designers get a set of current as well as future requirements, they can develop code structures that make future changes easy and cost effective compared to refactoring as needed. Designing with models is less expensive than designing in code.
- Starting development without a basic understanding of the project's vision and goals will likely lead to wasted effort as initial implementation will likely become useless over time.

Development teams should weight these arguments before deciding which approach they want to follow. In the following section, we will discuss techniques used by agile teams that are trying to strike a balance between these conflicting approaches.

2.3 Agile Project Planning

Agile teams usually plan on three levels of abstraction:

- Project vision
- Release plan
- Iteration plan

The project vision captures the really big picture: Why is the project run? What are the expected benefits? What are the budgetary and other constraints? How will the organization function after the project is successfully completed? A project vision is often used to establish a project budget or, at least, a budget that allows the organization to refine the vision enough so that a go/no-go decision can be made. Agile teams try to minimize this upfront work to avoid getting stuck in analysis without getting feedback about delivered product functionality.

A project vision needs to clearly describe the anticipated benefits for the business as well as assessment criteria that management can use to evaluate progress towards realizing the vision. Agile teams need management oversight to ensure that the next iteration/release still delivers enough business value to justify the development costs.

Release planning creates a strategic picture on the project. The team looks a few months ahead and determines the high-level features/user stories that need to be realized in that time frame. In practice, we observed teams creating release plans for the next three to six months, with a few exceptions looking approximately one year ahead. The release plan determines release dates and iteration length. Scope is captured on a high level but may be changed in the future based on new insights gained during development. User stories from release planning form the initial product backlog.

Iteration planning determines the work for the next development iteration. The whole team gets together to review what was delivered in the last iteration and then collaboratively determines what should be delivered by the end of the next iteration.

2.3.1 Planning a Release

For release planning, we recommend that the whole team gets together to

- collect and discuss high-level user stories that should go into the next software release,
- build a user story map, and
- create low fidelity prototypes.

Release planning is often conducted in a 1-3 day workshop involving the whole team and, if possible, external stakeholders. It starts with collecting user stories on different levels of abstraction: epics, themes, and implementation-ready stories.

*User Stories***.** A user story (also called: backlog entry or feature request) briefly describes a requirement that has business value. It serves as boundary object that enables communication between different stakeholder groups. According to Wikipedia, "A boundary object is a concept in sociology to describe information used in different ways by different communities. They are plastic, interpreted differently across communities but with enough immutable content to maintain integrity"[2].

A user story is captured on an index card (see Fig. 2). As a bare minimum, the user story has a name and a short description of the requirement. Descriptions need to be in customer language and avoid IT terminology. They need to be understandable by all team members. A user story also often includes effort estimates and is used to note actual effort during development.

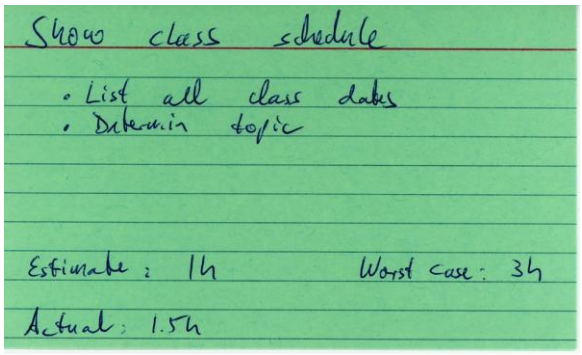

Fig. 2. Example story card

Mike Cohn, a prominent author focusing on agile project management, recommends a more structured approach for user stories:

- As a *[type of user]*
- I want to *[perform some task]*
- so that I can *[reach some goal]*

This structure helps business stakeholders prioritize user stories.

Index cards are small and will not be able to capture all details about the user story. They act as reminders to the developers to discuss these details with business representatives as soon as they start working on the story implementation.

Often, the back of the story is used to capture acceptance criteria for a user story. However, a more recent recommendation is to capture these in form of executable acceptance tests using frameworks such as Fit [9] GreenPepper[3] or BDD[4]. This approach is described in more detail in Section 3.

2.3.2 Mapping User Stories

[2] http://en.wikipedia.org/wiki/Boundary_object (last visited 29 July 2011)
[3] http://www.greenpeppersoftware.com/ (last visited 29 July 2011)
[4] http://dannorth.net/introducing-bdd/ (last visited 29 July 2011)

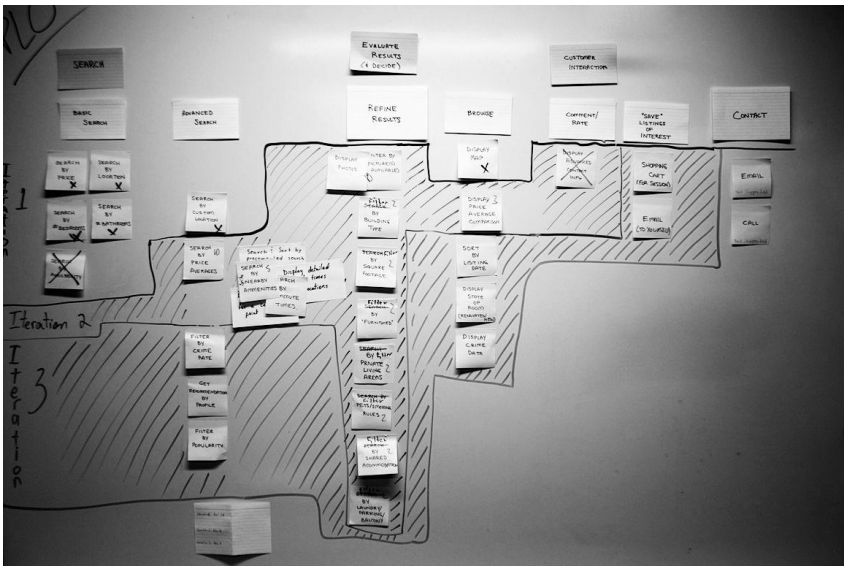

Fig. 3. Example user story map

A user story map[5] organizes and prioritizes user stories for a release. It makes the workflow of the system visible to the whole team and shows the relationship between large user stories and their parts. Fig. 3 shows an example user story map developed by a student team in its release planning workshop.

A user story map shows the sequence of activities of the system's workflow horizontally at the top of the board, left to right. The team then organizes (sub)tasks under their activity in the order of the workflow. This shows the relationship between activities and (sub)tasks while maintaining the time dimension of work steps. Concurrent or alternative tasks are added vertically by priority. Fig. 4 shows a conceptual example. Subtasks for tasks can be added as needed (not shown in Fig. 4).

[5] Developed by Jeff Patton
http://www.agileproductdesign.com/presentations/user_story_mapping/index.html (last visited 29 July 2011)

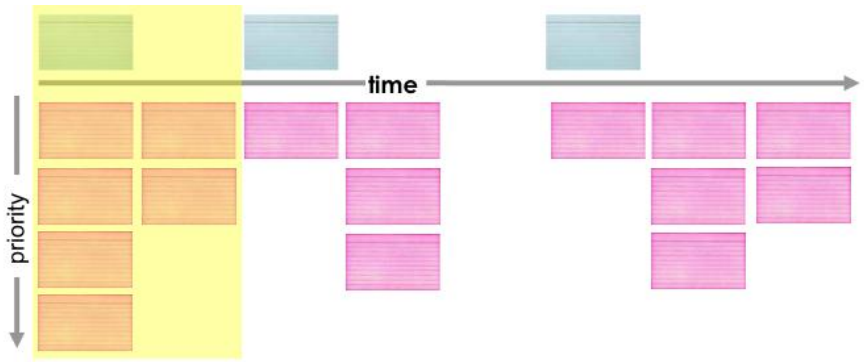

Fig. 4. Conceptual user story map

During release planning, the user story map is created, discussed, refined, extended, updated, and changed until it is complete. The team checks completeness by validating that the "story" of the system can be told by connecting the activity cards. It then evaluates if the tasks and subtasks provide enough information for answering more detailed questions.

After the user story map is complete, the team determines coherent sets of tasks for the iterations that make up the release. Fig. 3 shows this with an example.

The creation of a user story map is a – limited – upfront planning exercise. However, the time spent on it is substantially shorter than the time used for requirements analysis in more traditional processes.

When properly conducted, release planning is a collaborative exercise involving the whole team that creates a shared understanding of the release goals and system workflow.

2.3.3 Low-Fidelity Prototyping

User story maps allow a team to get a high-level overview of a system's feature set. A second technique that is used by agile teams to supplement the map is low-fidelity prototyping. Low-fidelity prototypes are particularly useful for development efforts with a strong focus on usability and interaction design issues. A low fidelity prototype is a sketch – a hand-drawn representation – of a user interface. A sequence of sketches that illustrates a workflow of a system is called a storyboard.

Teams use sketches and/or storyboards to capture the conceptual structure of the system's user interface. Fig. 5 shows an example sketch that illustrates the user interface of an agile project planning tool.

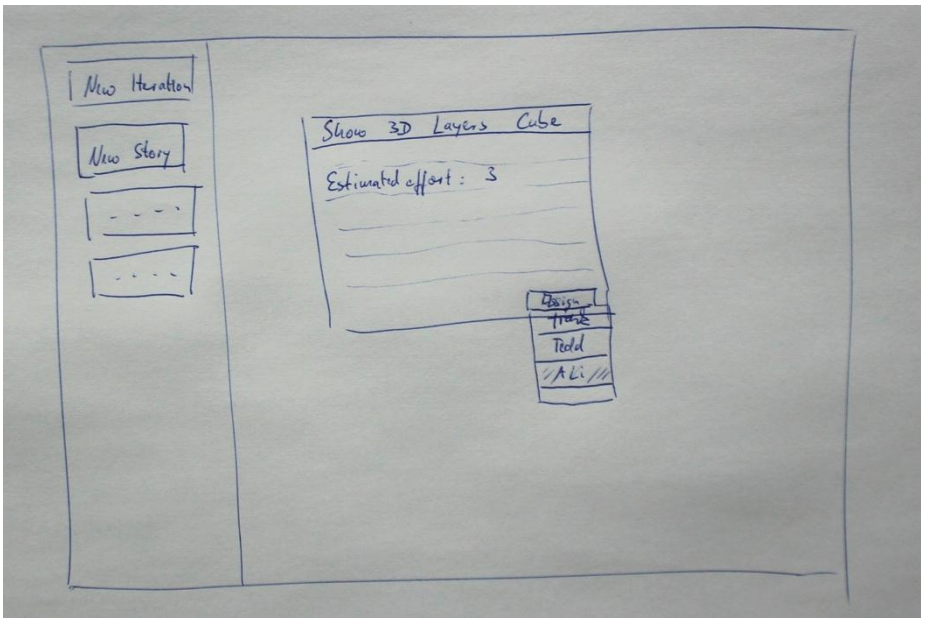

Fig. 5. User interface sketch

Bill Buxton's book [10] discusses the benefits of sketching for software development. Sketches are quick and easy to make. Thus, they fit well into short, iterative development cycles as exhibited by agile development teams. They can be provided when needed. As they are cheap to make, they are also disposable and their creator usually does not have a strong stake in them. As a result, teams often develop multiple alternative sketches for a user interface and discuss them with end users as well as other stakeholders. The series of ideas that is illustrated by alternative sketches often helps teams to clarify the design intent and the concept underlying the user interactions. The final system is often a combination of different ideas (expressed by different sketches).

Sketches also elicit feedback on the "right" level. Their rendering and style makes it clear that they are conceptual in nature and that the look of the system is not finalized yet. Users usually comment on conceptual structures instead of the choice of color or fonts. These conceptual level comments are exactly what interface designers need in the early stage of a development project.

Sketches help to start conversations between users and designers in the same way as user stories trigger discussions between developers and business representatives. Their value lies in provoking interactions between all stakeholders and helping teams derive better solutions for their customers.

Where a sketch illustrates the layout (or wireframe) of a single screen, a story board captures a workflow supported by a user interface. Story boarding is a technique borrowed from the movie industry: "Storyboards are graphic organizers such as a series of illustrations or images displayed in sequence for the purpose of pre-

visualizing a motion picture, animation, motion graphic or interactive media sequence, including website interactivity." [6]

Fig. 6 shows an example story board for an agile planning tool. The sequence of sketches illustrates how a user can create a story card, give it a name and effort estimate and, lastly, select the developer responsible for it.

Interaction designers can use sequences of sketches, as available from storyboards, to simulate the workflow with the user. These Wizard-of-Oz [11] experiments allow gathering feedback on the usability of a user interface before the implementation exists and are often used to ensure that even the first version of a UI creates a positive user experience.

In Section 3.3, we will discuss how automated Wizard-of-Oz tests based on storyboards can also be used for test-driven development of graphical user interfaces.

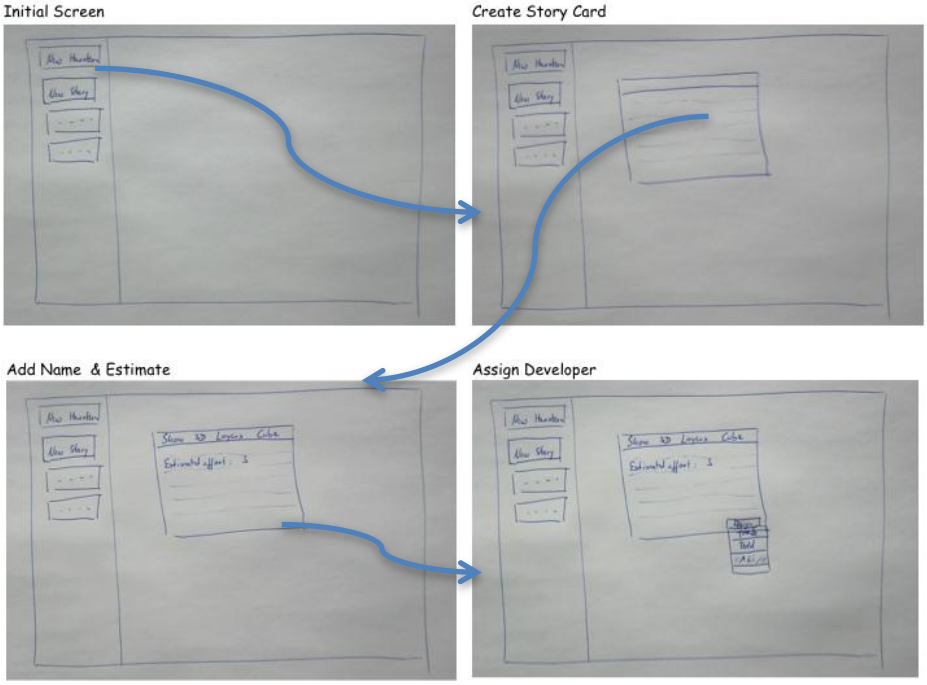

Fig. 6. Example story board

2.3.4 Iteration Planning

Iteration planning allows a team to get a tactical picture of the development effort: teams look ahead for a few weeks and create a realistic plan on what should be accomplished by the end of the iteration. The length of iterations is usually fixed in an agile project and the team will deliver on time (maybe with a reduced scope; see iteration planning). Most teams we work with run iterations for 2-4 weeks. Lately, there

[6] http://en.wikipedia.org/wiki/Storyboard

seems to be a tendency towards shorter iterations, i.e. we see more and more teams moving towards a two week cycle.

Following an iterative development process results in fixed delivery dates. Customers can expect to get a new set of production-ready features at the end of each iteration. Fixed delivery dates (with a slightly variable scope) have advantages:

- Sometimes external deadlines are hard. If a system needs to be demonstrated at a trade show to have a benefit for the developing company, the date is fixed externally and it doesn't help to deliver a more complete system after the show has ended.
- Reliable delivery of new features increases the trust of customers in the development team. Unfortunately in the past, customers were often burned by late deliveries and low quality. Thus, a constant and reliable delivery cycle increases the customer's confidence in the team and usually results in a more collaborative work environment.
- Developer motivation increases when they constantly deliver new features to their customers. Everybody likes to be successful – and delivering an increment is seen as a success.
- Putting new features into production allows the team to get feedback from actual use of the new functionality. While teams try their best to get everything right, the chances are that some details are wrong. Getting systems into actual use will quickly discover such issues and allow development teams to fix them quickly. Instead of accumulating technical debt over a long time, fast delivery will allow teams to deal with it in more manageable chunks.
- Reoccurring short-term delivery dates create some pressure on the team to focus their efforts on concrete steps. Parkinson's law states that work fills the time available for its completion. Short deadlines encourage teams to work on relevant tasks.

Iteration planning meetings normally run for a few hours and are attended by the whole team. The team selects the highest priority user stories from the user story map and discusses them. When needed, additional user stories are brought forward and the user story map is augmented accordingly.

The goal of team discussion is to enable developers to come up with a realistic estimate of the development effort for the story. These estimates together with the time available in the iteration allow the team to select a realistic set of stories that should be implemented in the upcoming iteration.

Cohn [12], p. 83+85, suggests that teams consider two dimensions when prioritizing user stories: business value and development risks. He suggests (see Fig. 7) to start with high risk, high value stories. Addressing high risk stories first allows a team to determine if the system is technically as well as economically feasible at all (and if not: cancel the project quickly before incurring the majority of the project costs).

Fig. 7. Business value and development risk

To determine how many user stories fit into the upcoming iteration, the team estimates user stories and determines its velocity.

Effort estimation: Effort estimates try to determine the size or complexity of a story by comparing it with others of similar complexity. Teams use different metrics for their estimates: (ideal) hours, story points or even gummy bears[7]. The goal of the estimates is to cluster stories that require similar efforts into the same bin – not to determine the amount of work hours needed for completing the user story (velocity is used for this). Typically, developers use their experience to determine an estimate. They remember similar tasks from the past and derive their estimate by remembering the effort of the past tasks. This means that estimates are mainly based on expert opinion and analogical reasoning.

Some teams use planning poker to derive estimates collaboratively. Each team member estimates for herself and then places a card with her best estimate on a table. If the set of cards shows different numbers from different developers, the team discusses these discrepancies and then estimates again until the estimates converge. Big discrepancies in estimates are treated as opportunities to refine the understanding of the story in the team as the differences are usually a result of an inconsistent understanding of what the story entails.

We recommend that developers provide two estimates for each story:

- Most likely estimate: the estimate that she thinks is really needed if no unexpected events happen while developing the story.
- Worst case estimate: the developer is asked to come up with a number that she is willing to guarantee

We treat the most likely estimate as a 50:50 chance that the actual effort needed to complete the story is at or below the estimate. On the other hand, we see the worst case estimate as a 95% chance that the actual effort is below the estimate.

Managers need to be careful in *not* treating most-likely estimates as commitments. The goal of estimation is to get the most realistic picture possible of what will happen in the next iteration. When estimates are treated as commitments or promises, developers will start over-estimating their effort to be on the safe side.

[7] The "gummy bear" metric attempts to make it clear that the number that is derived by the developers can not directly be mapped to calendar time.

Estimates are not 100% accurate. A team will only know how much effort a task is after it finishes working on it. Thus, planning is not about getting *the* correct picture but is about getting a perspective on the development project that allows a team to move forward while providing customers a good idea of what will be delivered at the end of the iteration.

For any iteration, estimates should stay within one order of magnitude. This prevents an effort overrun in one task from dominating the results of the iteration. When user story efforts are too far apart, small tasks can be combined or large tasks can be split. Splitting a task can be based on [12], p 121ff:

- the data supported by the story (e.g. Loan summary → List of individual loans → List of loans with error handling)
- operations performed within a story (e.g. separate create, read, update, delete (CRUD) operations)
- removing cross-cutting concerns (e.g. a story without and with security)
- separating functional from non-functional requirement (make it work, then make it fast)

When all user stories that might go into the next iteration are estimated, a team uses its velocity to determine how many of these are likely to be accomplished in the upcoming iteration.

Team velocity: A team's velocity determines how many story points are likely to be completed in the next iteration. Teams use a simple heuristic to determine this number: yesterday's weather. The assumption is that a team will be able to complete as many story points in the next iteration as it finished in the last iteration. The number is then slightly modified based on the number of person days in the upcoming iteration compared to the number of person days in the last one.

Combining story point estimates with velocity creates a simple approach for project planning. In our experience, it works rather well assuming that

- there are no major changes in the team and
- the team doesn't dramatically change its approach to estimating from one iteration to the next.

The approach is self-adaptive and corrects for developer optimism. A team that takes on too many user stories in one iteration will see its velocity reduced in the next iteration as they did not finish all their tasks. When a manager realizes that a team runs out of tasks in the current iteration, she can always go back to the business representatives and ask for more user stories. When they are also completed, the team's velocity will go up for the next iteration.

As estimates come from developers, some managers argue that they now have the power to slack off. However, this is counter-balanced by the customer's ability to cancel a project if progress is too small to accomplish its vision within a given budget.

While a team's velocity determines how many story points the business representatives can select for an iteration, one question remains: which of the two estimates should be used? The answer is: both. We usually recommend that teams first select a number of must-have stories for the next iteration based on the worst-case estimates. Business representatives can be quite sure that the developers will complete these tasks as the worst-case estimates will likely be met. However, the expectation is that

not all tasks will require the effort as determined by the worst-case estimate. Thus, the team selects as second set of optional user stories while keeping the sum of the most-likely case estimates of all selected stories below the velocity:

- $\sum_{i \in US_1} worst\_case\_estimate(user\ story_i) < velocity$
- $\sum_{i \in US} most\_likely\_case\_estimate(user\ story_i) < velocity$
  where $US_1$is the set of must have stories, US_2 is the set of optional user stories and $US = US_1 \cup US_2$

These constraints on the one hand ensure that the customers know at the beginning of the iteration which user stories will definitely be delivered while any remaining time is filled with optional user stories based on the customer's priorities.

2.4 Progress Tracking

Agile teams track their progress on three levels of abstraction:

- Daily: Are we in trouble at the moment?
- Iteration: Will we make our tactical goals?
- Project: Will we reach our vision?

For tracking daily progress, most agile teams use a short stand-up meeting at a regular time. During the daily stand-up, each team member reports on three questions:

- What have you done since the last meeting?
- What will you do before the next meeting?
- What is in your way?

The meeting is limited to at most 15 minutes and held at the same time and place every workday. The meeting is not meant for problem solving but for bringing issues to the attention of the whole team so that an appropriate group of people can be identified that can get together after the stand-up and find a solution.

Nobody sits during a stand-up. This encourages people to keep everything short.

Daily stand-ups force people to think about their short term goals and report on their short term accomplishments. The latter creates some benevolent peer-pressure as developers not making any progress on their tasks for several days in a row become very visible. The last question addressed by each team member helps to discover roadblocks quickly. The earlier a team knows about an issue, the earlier it can find a solution.

Tracking progress within an iteration is done with task boards. A task board shows the stages through which each user story/task goes and where it currently is. Fig. 8 shows an example task board from Mike Cohn's web site. In this example, user stories are split into individual tasks. These tasks go through four stages: to do, in progress, to verify and done. Each row in the task board shows the tasks for a certain user story. Task boards are widely used by agile teams. They act as widely visible information radiators that help all team members to understand how much progress is being made in the current iteration.

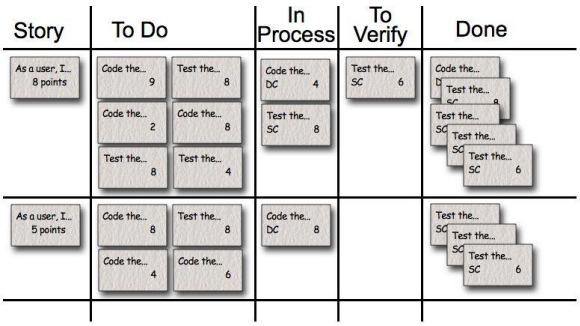

Fig. 8. Task board example[8]

For a more detailed tracking of progress against the iteration goals, teams sometimes use burn-down charts. These chart the amount of not-yet completed tasks on a daily basis [2].

At the end of each iteration, an iteration review is conducted to show to product owners and customers/users how much progress was made during the iteration. During the review, the team demonstrates all features that were completed in the current iteration. An iteration review should not impose extra overhead on the development team. Thus, it is conducted using the development equipment. Features shown during the review must represent potentially shippable product functionality: i.e. it is then a business decision if the feature goes live or not.

Iteration reviews give senior management an opportunity to see if the progress that was made is sufficient to make the project vision reachable within the given budgetary and scheduling constraints.

2.5 Business Contracts

Historically, software development contracts had a time-and-expenses structure, i.e. customers paid a fixed amount of money per developer hour plus reimbursed the team for all expenses incurred by the project. However, given the amount of cost overruns in the past, customer organizations switched to a fixed price/fixed scope structure: customers pay a fixed amount of money to the development team that in turn must deliver a set of features laid out in a detailed requirements specification. This approach tries to move the risk of incorrect estimates from the customer side to the developer side. Unfortunately, the success of this approach is quite limited:

[8] From http://www.mountaingoatsoftware.com/scrum/task-boards

- Development organizations realize that a fixed price/fixed scope contract makes them vulnerable for incorrect effort estimates. After they determine the honest effort estimate for a contract, they will use a risk multiplier when they submit a bid. This multiplier de-facto serves as an insurance premium that a client has to pay to the development organization to assume the technical risk of the contract. If the initial effort estimate is correct, the client organization pays more than needed to avoid the risk of becoming burned by incorrect estimates.
- The size of the risk multiplier is determined primarily by how urgently the development organization wants the contract. When business is booming, customers will pay a high premium. When it is slow, they will pay less. However, even when a development organization lowballs the project bid to get the contract, the customer still will pay too much as developers know that over the course of a project customers always change the requirements.[9] Developers can overcharge for changes as switching the development organization mid-project is usually not economically feasible for the customer organization due to penalties written into the contract. Fig. 9. illustrates these issues.

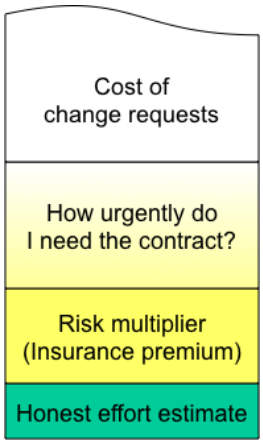

Fig. 9. Budget determination

Agile organizations can replace fixed price/fixed scope contract with a time-and-expenses contract with an early termination clause. The later limits the risk of the customer organization as it can cancel the project quickly when it realizes that the project will not be able to deliver on its vision given the current budget constraints.

While project planning and progress tracking are important aspects of agile software development processes, bad software delivered on time is still bad software. Thus, we are now discussing how agile teams assure that they delivered high quality.

3 Agile Quality Assurance

Have you ever worried that the feature you've been developing doesn't match the expectations of your customer? Have you ever been reluctant to change code because you might break something? Have you ever been unsure about whether or not you've finished a feature? Have you ever been terrified that one of the other developers might go on vacation, and that no one else will be able to understand what his code does?

Agile quality assurance is a set of testing methodologies that have evolved over time to minimize these risks on software development projects. The overall goal of these methodologies is to increase understanding of and communication about the system that's being developed. These practices can be divided into two classes: de-

[9] We've been involved in software development for about 30 years now and haven't seen a single project where requirements stayed fixed for its whole duration.

veloper-facing and customer-facing tests. Tests written by developers ultimately help them design and understand the system, while tests written under the auspices of customers help developers understand what customers want and help customers understand what developers can offer.

Developer-facing tests require in-depth knowledge of the way in which the system works, and require technical proficiency in a testing language to understand. These tests are usually glass-box (or white-box) tests in which parts of the source code of the system are tested. This name derives from the fact that the system under development is being treated as something we can look into and inspect the intermediate results of our actions. For example, in glass-box testing, we can set or inspect the state of specific objects or call only specific methods within the application rather – as opposed to triggering high-level functionality, which would result in changes to the states of many objects and many method calls. This allows much more fine-grained understanding of the way in which a system works, and will help developers ensure that the feature they are coding matches their goals for its behavior – in other words, that developers are building the system right. Developer-facing tests are almost always automated through a testing framework like JUnit[10] or the Visual Studio Unit Testing Framework[11]. Examples of developer-facing tests include unit tests, integration tests, and system tests.

Customer-facing tests, on the other hand, are intended to be understandable by domain experts without requiring programming knowledge. These tests tend to be black-box tests in which the internals of the system are not considered. An input is provided, and the expectations of the business experts are compared against the output the system produces. This sort of test ensures that the code created by developers fulfills customer expectations – in other words, that developers are building the right system. Customer-facing tests can be automated (through a system like FitNesse[12] or GreenPepper[13]) or manual (through live demos on the actual system).

Customer- and developer-facing tests can be envisioned as two partially-overlapping squares, as shown in Fig. 10. Developer-facing tests can show that individual parts of an application are working in detail on a programmatic level, but not that defined features are missing. Customer-facing tests, on the other hand, can show that features are present, but not that they are working in detail on a programmatic level.

[10] http://junit.org
[11] http://msdn.microsoft.com/en-us/library/ms243147.aspx
[12] http://fitnesse.org/
[13] http://www.greenpeppersoftware.com

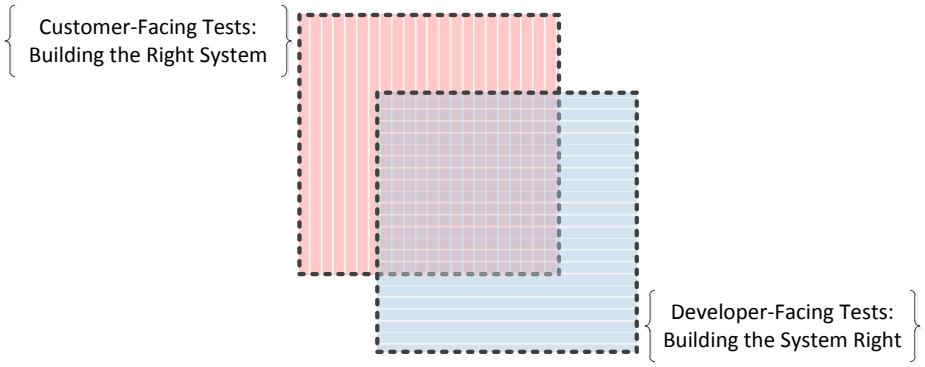

Fig. 10. Both genres of tests are necessary in agile quality assurance

Finally, agile quality assurance tends to make heavy use of automated developer- and customer-facing tests. This is due to the fact that the same tests will tend to get run a large number of times on an agile project. For example, refactoring is a key concept in agile software development. However, there is a risk that developers may introduce errors into the program while performing this task. A suite of automated tests can catch these errors quickly, which both emboldens developers to aggressively refactor their code while at the same time making the refactoring process much safer. In this light, automated tests are definitely worthwhile.

This isn't always the case though – it's a fallacy to think that you have you automate every single test on your project. This is because some automated tests actually cost more to create and maintain over the course of the project than a manual equivalent. Brian Marick addressed this point eloquently in 1998:

"It took me a long time, but I finally realized that I was over-automating, that only some of the tests I created should be automated. Some of the tests I was automating not only did not find bugs when they were rerun, they had no significant prospect of doing so. Automating them was not a rational decision" [13].

Remember, the point of test automation is to save effort in the long term. If a test is difficult to automate or doesn't have a reasonable chance of catching bugs, it may be more cost-effective to run this test manually. The best tests to automate are those that have a good chance to find bugs over a long life expectancy.

3.1 Test-Driven Development

Test-driven development (TDD) is a software development paradigm in which tests are written before the code they are referencing actually exists. The tests used in TDD are assumed to be automated, developer-facing unit tests unless otherwise specified. This activity is more about software design and communication than it is about testing per se, though it does build up a suite of regression tests that are useful for detecting errors introduced by changes made later on. The goal of TDD is to increase the confidence that developers have in their code, decrease the occurrence of bugs that make it

through to the customer, prevent the re-introduction of bugs, and increase communication between developers and customers.

The first step in TDD is to write a new test. This test should be confined to a single new part of the system – a new method, a new class, or a new feature depending on the scope of the testing. This causes the system to enter a red state: at least one test is failing. In other words, there is something wrong with the system - it's missing the part specified by the new test. This defines a goal for the developer: get the system back to a working state as quickly as possible. From this perspective, tests are driving the development of the system.

Initially, this new test should be the only failing test for the system, so the next step is to verify that this test is failing. If this new test passes immediately upon creation, either:

1) there's something wrong with the test; or
2) the "new" part of the system already exists, meaning no new code is necessary; or
3) the developer misunderstood the current design of the system as a test that is expected to fail in fact passes, meaning the developer will have to increase her knowledge about the system.

Once we've watched our test fail, code should be written with the specific goal of getting the new test to pass - no code should be written unless it directly relates to making this test pass! Additionally, it's alright if our code is not perfect at this point, because we'll improve it in the next step.

Once the test is passing, the system is back in a green state (all tests are passing), and we can focus on the crucial last step: refactoring. In the previous paragraphs, the emphasis was on speed. This means that it's crucial for us to go back to the new code to make it efficient, secure, robust, maintainable, or any one of a number of software quality concerns. However, this process is a safe one now because of the new test. If our refactoring causes this test – or any other test – to fail, our first priority again becomes getting the system back to a working state. Because of the suite of regression tests built up through TDD, developers can aggressively refactor the code base of an application.

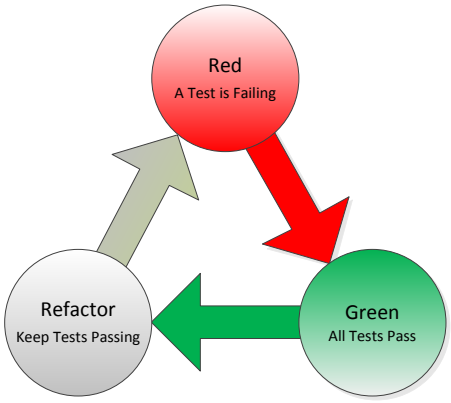

Fig. 11. The test-driven development cycle

Evaluations of TDD have had mixed results. In general, it would seem that TDD has a negative effect on productivity and a positive effect on quality [14]. However, as was mentioned in Section 2.2, bugs found after release of an application are significantly more expensive to address, so any decrease in productivity needs to be viewed with this in mind as the studies in the above mentioned publication did usually not include data about post-deployment productivity comparisons.

3.2 Acceptance Test-Driven Development

In Acceptance Test-Driven Development (ATDD), instead of creating automated, developer-facing unit tests, we create a suite of customer-facing system tests. These tests are created before the features they test are implemented, as in TDD. However, in ATDD, these tests should actually be created by customer representatives as descriptions of what the application should behave like when it is working correctly. Because of this, many acceptance testing frameworks, like FitNesse and GreenPepper, include an interface that is friendlier to non-technical test writers. In practice, business representatives may still need assistance in writing tests, in which case they should be paired with testers who can help them write tests (not write tests for them!). An example of a FitNesse test (showing a hypothetical business expert's expectations for the result (right) of division given a specific numerator (left) and denominator (middle)) is shown in Fig. 12.

```
|eg.Division|
|numerator|denominator|quotient?|
|10       |2          |5        |
|12.6     |3          |4.2      |
|100      |4          |33       |
```

Fig. 12. Example FitNesse acceptance test[14]

It's important to note that most tools which are advertised as acceptance testing tools interact with an application below the level of its user interface by directly making calls into business methods of the application and verifying the results. This is useful in that avoiding the GUI simplifies the creation of an automated test drastically. However, if parts of the GUI are important to a customer's acceptance criteria, it will be difficult to automate that part of the test for ATDD using such tools. It's possible to write acceptance tests that involve interactions with a GUI, but few methods exist that make it easy to perform ATDD of a GUI.

As with TDD, tests created for ATDD should be automated wherever possible. Acceptance tests written using many acceptance testing tools can be run alongside other automated tests as part of the suite of regression tests. This means that changes to the application under test that cause violations of the customer's acceptance criteria can be detected quickly and easily.

However, the functionality of modern applications is becoming increasingly difficult to test automatically. This is due in part to the fact that modern applications are heavily dependent on GUI-based interactions. While it is easy to automate tests of the functionality of a standard webpage or desktop application using FitNesse tests, it's difficult to automate tests of applications with complex GUIs. In these instances, it may be preferable to specify manual tests using a tool like Microsoft Test Manager[15], which integrates manual tests with other software development tools. When using manual tests as part of ATDD, however, developers will need to execute these tests manually numerous times during development, which can be a tedious and expensive process.

3.3 Test-Driven Development of Graphical User Interfaces

User interfaces are an important part of almost every modern application. Simply put, they allow users to interact with applications. Traditionally, this was done with keyboard and mouse, but this is now possible using touch input in mobile phones (like the iPhone and Windows Phone 7), tablet computers (like the iPad and Asus EEE Slate), and digital surfaces (like the Microsoft Surface, SMART Board, and SMART Table). Further, up and coming technologies like the Microsoft Kinect are making it

[14] Source: http://fitnesse.org/FitNesse.UserGuide.TwoMinuteExample

[15] http://msdn.microsoft.com/en-us/VS2010TrainingCourse_AuthoringAndRunningManualTests

possible to interact with a computer without even touching it. Clearly, user interfaces are an important and complex concern in software development.

Further, user interfaces can be either event-driven or loop-driven. Event-driven interfaces primarily respond to input from the user. Examples of event-driven interfaces include traditional desktop applications and web pages. Loop-driven interfaces are primarily driven by the passage of time, but will also take user input into account. Many computer games are excellent examples of loop-driven interfaces. The difference is that in an event-driven interface a sequence of interactions will produce the same result regardless of timing, but in a loop-driven interface this is unpredictable.

For the purposes of this chapter, let us consider only event-driven graphical user interfaces (GUIs) based on mouse, keyboard, or touch interaction. While powerful patterns for dealing with the complexity of GUIs exist (e.g. the Model-View-Controller pattern), there is still a significant amount of code present in a GUI – in fact, 45-60% of an application's code can be dedicated to its GUI [15]. In line with this, one case study found that 60% of software defects found after release relate to GUI code, and of these defects, 65% result in a loss of functionality [16]. Taken together, these studies suggest that GUI testing is an area of significant concern.

However, automated GUI testing is far from straightforward. In order to better understand what makes GUI testing a daunting task, let us consider four fundamental concerns of automated software testing made especially clear in this context:

- Complexity,
- Verification,
- Change, and
- Cross-Process Testing

The complexity of an application refers to the number of alternative actions that are possible. GUIs allow a great amount of freedom to user interaction, making them very complex. When testing the functionality of a GUI-based application using automated tests, two factors are of prime importance: the number of steps in the test; and the number of times each action in the GUI is taken within a test [17]. In Section 3.4, the implications of the complexity of modern GUIs will be explored in more detail. In order to notice that a bug has been triggered, a test must also contain verifications that will be able to notice that bug [18]. This is especially tricky when considering that many aspects of GUIs are subjective. For example, it can be difficult to create a test for determining whether a web page was rendered correctly. Third, GUIs tend to change drastically over the course of development. A GUI test can show up as failing although the underlying code is actually working [19] [20]. This is especially important since a large number of false alarms from the GUI testing suite will cause developers to lose confidence in their regression suite [21]. Finally, these difficulties are compounded by the fact that GUI tests generally interact with a GUI from a different process. This means that the test will not have access to the internals of the GUI it is testing. Instead of simply calling a method on an object, it's necessary to first locate that object within the GUI. This is generally done by traversing the tree of graphical elements from the root window object until a widget matching details of the desired widget – as it appeared when the test was created – is found. Additionally, because of this cross-process testing, it's rare for all information about a widget to be

exposed. For example, the Button class as implemented in Windows Presentation Foundation[16] can be tested through the InvokePattern interface in the Windows Automation API[17]. The Button itself has 136 properties, but InvokePattern exposes only 20 of these for use by test code. It can be difficult to create strong tests in the (common) case that one of the properties that isn't exposed is important to the functionality of a feature.

With these concerns in mind, we need to consider what the purpose of our GUI testing actually is. There are two distinct forms of GUI testing: testing the look of the GUI; and performing system testing of the application through its GUI. Take for example the Wikipedia entry for GUI testing, Fig. 13. If we want to verify that on this page that the Wikipedia logo appears as the upper-leftmost widget, that directly below it is a "Main page" hyperlink, and so on, we are testing the look of the GUI. If instead we want to verify that clicking on the link to software engineering in the first paragraph takes us to a page titled "Software engineering – Wikipedia, the free encyclopedia," then we are testing the functionality of the system as a whole. Again, for the present, let us consider the second of these approaches. Essentially, we are performing ATDD through the application's GUI instead of below it.

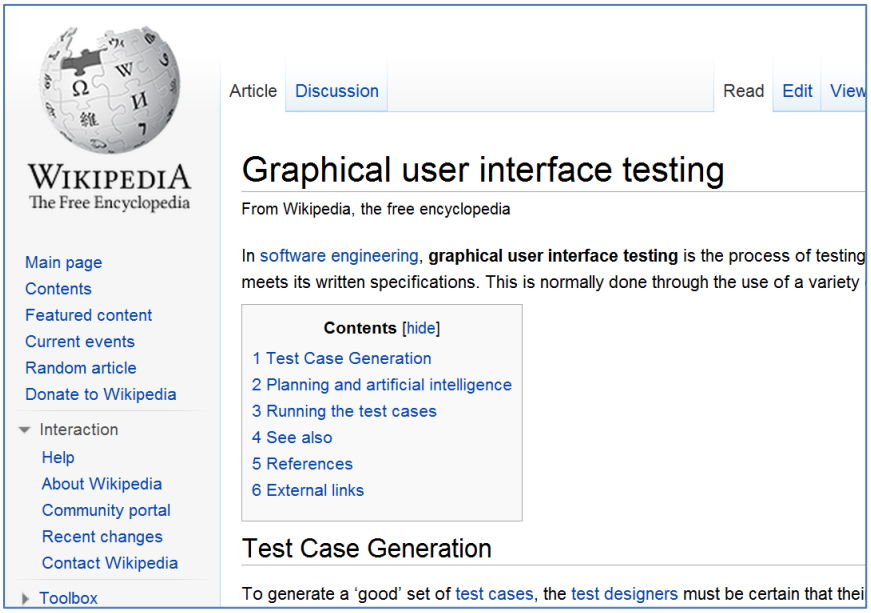

Fig. 13. A sample GUI[18]

[16] http://msdn.microsoft.com/en-us/library/ms754130.aspx
[17] http://msdn.microsoft.com/en-us/library/dd561932(v=VS.85).aspx
[18] From http://en.wikipedia.org/wiki/Graphical_user_interface_testing

It is possible to write GUI tests for use in test-driven development of a GUI manually using available GUI testing tools. For example, it is entirely possible to write a Selenium[19] test by hand before a GUI exists even though Selenium is primarily a capture/replay tool (CRT) – a testing application that records a series of interactions with a system and records them in a format that can be replayed later as a test. This approach has been supported in the past by tool like TestNG-Abbot [22] and FEST [23], but has not received widespread uptake. This could be due to the fact that test authors need to know a large amount of detailed information about the GUI to be created in order to write a test.

A simpler approach to UITDD involves the creation of an automated low-fidelity prototype using a program like ActiveStory Enhanced [24] or SketchFlow[20]. These prototypes are event-based GUIs that respond to user input in the same way in which actual GUIs do. This means that they generate events when a user interacts with them. These events can be captured using a CRT, like white[21] or LEET [25], in the same way in which they can be used to record events from an actual GUI. These events can then be replayed on the actual GUI, with one caveat: the elements in the prototype that are generating events need to have the same identifying information as the equivalent elements in the actual GUI.

Consider for example the prototype shown in Fig. 14. It was created in SketchFlow, which means that each widget will raise recordable events when interacted with. We can use this prototype both for testing the actual GUI and testing the actual application through its GUI. From the prototype, we can use information about, for example, the arrangement of widgets to create tests of the GUI, or we could use the functionality demonstrated through the prototype to create acceptance tests of the actual application. For example, we can fill in the fields as shown in Fig. 14, then click the "Clear Report" button and verify that the fields have been cleared. We can then use this test for verification of both the form and functionality of the actual application, Fig. 15.

[19] http://www.seleniumhq.org
[20] http://www.microsoft.com/expression/products/sketchflow_overview.aspx
[21] http://white.codeplex.com

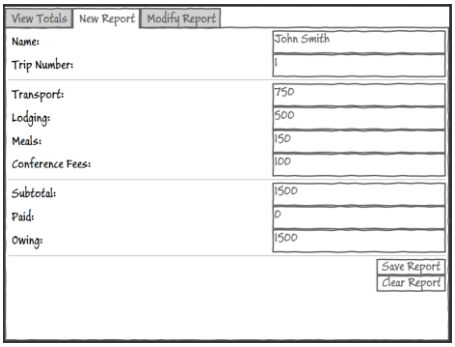

Fig. 14. Prototype of Expense-Manager's GUI

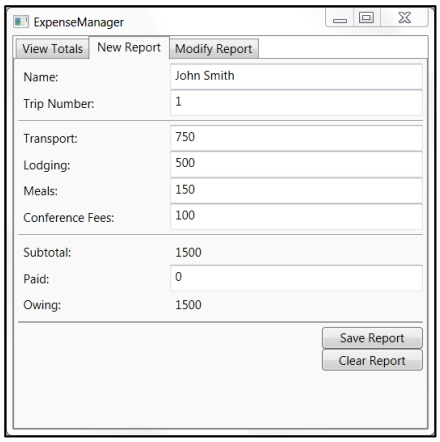

Fig. 15. Implementation of the GUI of ExpenseManager

There are several advantages to this approach. First, this approach to UITDD has the advantage of being able to make use of CRTs, which makes it much easier to create tests than it would be to create them by hand. Second, by integrating medium-fidelity prototyping into the TDD process, we are creating another opportunity for testing – usability testing, as described in Section 2.3.3. This means that it is possible to detect usability errors early in the development process, which not only makes them cheaper to fix, but also reduces the number of changes to the GUI that will be necessary later in the software development process. This reduces the risk that changes to the GUI will break GUI tests since there will be fewer of them. Third, this approach reduces the apparent complexity of the application we're testing by helping us specify which parts of it are going to be important early on. Only the important flows of each feature of the application will be shown in a low-fidelity prototype, so we will automatically know which parts of the application (and which sequences of events) we need to focus on when we are recording tests.

3.4 State Space of Testing

The state of a program is the set of values of all variables defined and instantiated by that program. The state of the system grows when new objects are instantiated and shrinks when garbage collection takes place. In an object-oriented application, the state should be represented as a graph with each node containing a set of objects, each of which also contains the state of each of its fields. The state space of an application, then, is the graph of all possible states that the application can enter. Method calls cause the application to transition from one state to another by changing the values of variables. In a completely deterministic program, the state space would be a linear sequence of states leading to a single terminal state. Whenever a program can be influenced by outside factors – such as interaction with the file system, input from a user, or any number of other events – then the state space will become a graph with

multiple edges between many of its state nodes. The more possible states there are, the larger the state space becomes.

For example, consider Fig. 16. In this example, the states of the system are shown within square brackets with highlight boxes surrounding new information and method calls that cause transitions are described within callouts to the numbered arrows. Note that each method call has a discrete – and testable – effect on the overall state of the system.

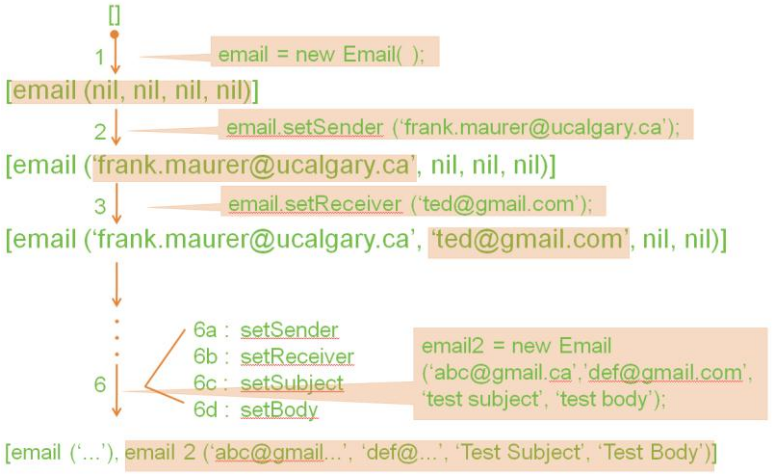

Fig. 16. A subset of the state space of a sample email application

When viewing a system as a state space to be explored, the goal of a software tester is threefold:

- To map out and understand the system
- To set up automated tests that will detect new bugs introduced by changes
- To search for new bugs in unexplored regions of the state space

The difficult with GUI testing from this perspective is: how can our tests cover the important parts of the state space effectively when we can only create so many tests? This concern will be addressed in more detail in the following section.

There are many issues to consider when viewing an application under test as a state space to explore. First, what starting state should be used? Many applications, for example web sites or document viewers, can be accessed initially in many different ways. Second, the state of an application may not be entirely visible to test code. For example, with black-box GUI tests, it's possible to make verifications regarding the state of the GUI, but not about the state of the underlying application. This makes it difficult to actually determine whether features are entirely working, or even if a test was able to navigate through the state space to the correct state. Third, as an extension of this last point, an application's state space should be viewed as a subset of the state space of the computer as a whole. Interactions with the file system, network, or a database, processing and threading timings controlled by the operating system, and even time can be an important part of the state of an application. Finally, once we've chosen a starting point, how do we move the system into a state in which we are in-

terested? For some applications, like websites, it's possible to navigate directly to a desired state. For other GUI-based applications, there may be only one starting state, and there may be many intervening states between the starting state and a state that we want to test. Changes to these intervening states can cause a test to fail when the functionality the test was initially intended to test is still working. This is because the test isn't able to navigate to the correct state. In such a case, either the test will generate an exception by trying to perform impossible actions on the GUI, or verifications will fail because they are being run on a different state then they were intended to. Both of these failures can occur when the system is actually working correctly. An illustration of this can be found in Fig. 17.

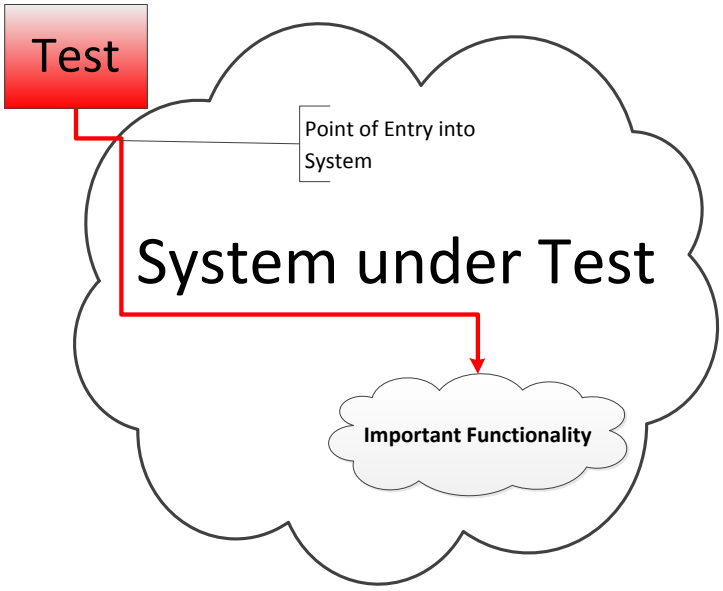

Fig. 17. Tests may need to traverse significant portions of the state space to reach and test interesting functionality

3.5 Test Quality

There are a variety of methods available to determine how good our testing is. Two of the most popular are code coverage and mutation testing.

Code coverage is a measure of how much of the system a test suite actually enters during testing. However, there are many different types of code coverage. The most lenient definition is line coverage (also known as: statement coverage). When code coverage is referred to in an agile environment without specifying what kind of coverage metric is being used, this is the type that is meant. Line coverage is a measure of the number of lines of the application that were executed during a test run, but doesn't account for the quality of that execution. For example, consider an "if" statement that can resolve in two distinct ways. From the perspective of line coverage, it doesn't matter which way the condition is resolved – the if statement itself will be considered

covered either way. Close to the other extreme, we have multiple condition / decision coverage (MC/DC). In MC/DC,

- Every result of every decision (set of conditions) must be evaluated,
- Every result of every condition must be evaluated,
- Each condition in a decision must be shown to independently affect the result, and
- Each entry and exit point of the program must be used.

This method of evaluating code coverage is very exact, and is used in instances where a software failure would have catastrophic consequences – such as software used in guidance and control of aircraft. As with all code coverage metrics, the goal is to get as many states within the application's state space as possible visited by test code, if not verified in detail.

Mutation testing, on the other hand, is a very different approach to checking the quality of a test suite. In mutation testing, we actually modify parts of the system, then run our tests against this "mutant." If our tests are not strong enough to realize that the system has changed, then the tests need to be modified. By combining mutation testing and code coverage, we can get information not only about which parts of our system haven't been tested yet, but also when our verifications about a state aren't strong enough.

3.6 Testing Graphical User Interfaces – a State Space Explosion

As was alluded to in the previous section, state spaces tend to be very large. This is especially true when we consider the state space of a GUI-based application. Consider, for example, the primitive calculator application shown in Fig. 18. How many visible widgets does it contain? How many properties do you think are contained in all of those widgets, and how many methods are there to call?

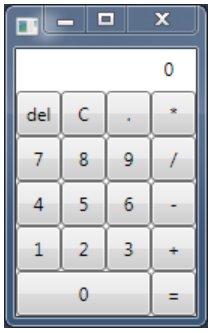

Fig. 18. A simple calculator application.

In this application, there are:

- 19 visible widgets,
- 2577 properties of these widgets, and
- 4007 methods that can be called on these widgets.

In this supposedly-simple application, there are thousands of properties that go into the definition of **each** state that the system can enter. This means that it would not be possible attempt to visit every state of the application and verify every property. Considering the fact that GUI errors do impact customers, and that GUIs for applications that are used in the real world are significantly more complicated than this example, it's extremely important to think about GUI testing in terms of exploring the state space of an application in a way that will provide a good return on investment for our effort.

So, how do we deal with the fact that it's easy to create systems which will be impossible to conclusively test? We have to prioritize which parts of the state space are more likely to contain bugs or are susceptible to the future introduction of bugs. These parts of the state space are more important than others – for example, division by zero is a significantly important concept in our calculator application, and every attempt to divide by zero from any state should result in a transition to the same state (a state in which "cannot divide by zero" or the equivalent is displayed).

The corollary to this is that, where we can identify parts of a system that introduce *unnecessary* complication into the state space, we can encapsulate this complexity using mocking (sometimes referred to as isolation). That is, we can encapsulate complexity that is not essential to the purpose of a test so that our tests are more reliable, simpler, and have fewer dependencies. Mocking frameworks, such as Moles[22] or jMock[23], allow us to detect when a complex object would be created and instead replace it with a mock object. Mock objects can be interacted with by other objects in the same way as the object they are replacing. However, they will return a predetermined value instead of interacting with other parts of the system. Additionally, mock objects can record the parameters used in method calls into their methods so that we can verify later on that other parts of the system are interacting with the object we are mocking in an appropriate manner. In essence, this allows us to reduce the size of in individual state or of the state space as a whole by replacing a set of complicated, difficult-to-verify objects or chains of method calls with a single, predictable object or state. An illustration of the latter is provided in Fig. 18.

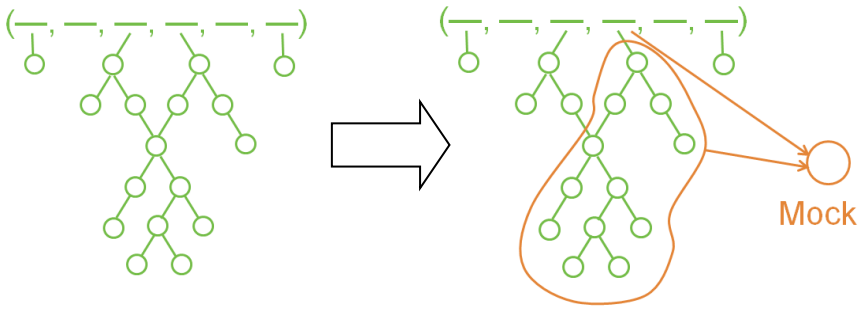

Fig. 18. Replacing part of the state space with a simple mock.

[22] http://msdn.microsoft.com/en-us/library/ff798506.aspx
[23] http://jmock.org/

For example, when a feature interacts with a database, the state space of the database becomes part of the state space that we are testing. This can slow down test execution and cause confusing test failures for a host of reasons that are completely unrelated to the purpose of the test: make sure the feature is working. Testing the feature inclusive of the database additionally tests the network connection to the database, the database itself, the database contents etc. Rather than putting up with this additional complexity, we can simply mock out the portion of the system that relies directly on this database and instead work with predefined, predictable data. Mocking can be used to avoid a range of complications that regularly complicate testing, from database and networking issues to file access to testing features that depend on a specific date to testing multi-threaded applications. This allows us to focus on the real question: given that all its dependencies are working, does our feature work correctly?

4 Summary and Conclusions

This chapter gave an overview of core strategies used by agile software development teams.

We focused our discussion on two aspects: project management and quality assurance. Fig, 19 summarizes our discussion. It shows that agile teams use an iterative development processes with a daily feedback loop consisting of standup meetings. User story maps and low fidelity prototypes are used as cost-effective means of capturing the strategic view of the projects. These are updated based on new knowledge gained in the current iteration. The development team primarily delivers source code and tests. The team demonstrates the iteration results at the end of each development cycle to all stakeholders. This in turn is the basis for the next iteration planning meeting, which determines the goals for the upcoming cycle.

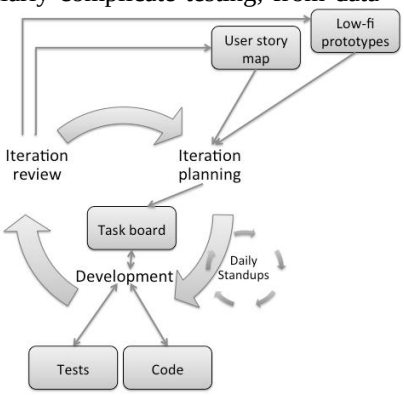

Fig. 19. Development process

Agile processes are mainstream software development methodologies used by many teams within the software development industry. While they are no silver bullet and require substantial rigor and commitment from teams, they seem to be delivering results.

References

1. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (1999)
2. Schwaber, K.: Agile Project Management with Scrum. Microsoft Press (2004)
3. Poppendieck M., Poppendieck, T.: Lean Software Development: An Agile Toolkit. Addison-Wesley (2003)
4. Palmer, S.R., Felsing, J.M.: A Practical Guide to Feature-Driven Development. Prentice Hall (2002)
5. DSDM Consortium: DSDM: Business Focused Development. Jennifer Stapleton, (ed.) Pearson Education (2003)
6. Cockburn, A.: Crystal Clear: A Human-Powered Methodology for Small Teams. Addison-Wesley (2004)
7. Highsmith, J.A.: Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. Dorset House (1999)
8. Pressman, R.S.: Software Engineering: A Practitioner's Approach. McGraw-Hill, Boston (2001)
9. Mugridge, R., Cunningham, W.: Fit for Developing Software: Framework for Integrated Tests. Prentice Hall (2005)
10. Buxton, B.: Sketching User Experiences: Getting the Design Right and the Right Design. Morgan Kaufmann (2007)
11. Kelley, J.F.: An Iterative Design Methodology for User-Friendly Natural Language Office Information Applications. ACM Transactions on Office Information Systems, vol. 2, no. 1, pp. 26-41. ACM (1984)
12. Cohn, M.: Agile Estimating and Planning. Prentice Hall (2005)
13. Marick, B.: When Should a Test Be Automated? Proceedings of the 11th International Software Quality Week, vol. 11, San Francisco (1998)
14. Jeffries, R., Melnik, G.: Guest Editors' Introduction: TDD - The Art of Fearless Programming. IEEE Software, 24-30 (2007)
15. Memon, A.M.: A Comprehensive Framework for Testing Graphical User Interfaces. PhD thesis, University of Pittsburgh (2001)
16. Robinson, B., Brooks, P.: An Initial Study of Customer-Reported GUI Defects. In: IEEE International Conference on Software Testing, Verification, and Validation Workshops, pp. 267-274. IEEE (2009)
17. Xie, Q., Memon, A.M.: Using a Pilot Study to Derive a GUI Model for Automated Testing. In: ACM Transactions on Software Engineering and Methodology, vol. 18, no. 2, pp. 1-35. ACM (2008)
18. Memon, A., Banerjee, I., Nagarajan, A.: What Test Oracle Should I Use for Effective GUI Testing? In: 18th IEEE International Conference on Automated Software Engineering, pp. 164-173. IEEE (2003)
19. Memon, A.M., Soffa, M.L.: Regression Testing of GUIs. In: ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 118-127. ACM (2003)
20. Memon, A.M.: Automatically Repairing Event Sequence-Based GUI Test Suites for Regression Testing. In: ACM Transactions on Software Engineering and Methodology, vol. 18, no. 2, pp. 1-36. ACM (2008)

21. Holmes, A., Kellogg, M.: Automating Functional Tests Using Selenium. In: AGILE 2006, pp. 270-275. IEEE (2006)
22. Ruiz, A., Price, Y.W.: Test-Driven GUI Development with TestNG and Abbot. In: IEEE Software, pp. 51-57. IEEE (2007)
23. Ruiz, A., Price, Y.W.: GUI Testing Made Easy. In: Testing: Academic and Industrial Conference - Practice and Research Techniques, pp. 99-103. IEEE (2008)
24. Hosseini-Khayat, A., Hellmann, T.D, Maurer, F.: Distributed and Automated Usability Testing of Low-Fidelity Prototypes. In: International Conference on Agile Methods in Software Development, pp. 59-66. IEEE (2010)
25. Hellmann, T.D., Maurer, F.: Rule-Based Exploratory Testing of Graphical User Interfaces. In: International Conference on Agile Methods in Software Development, pp. 107-116. IEEE (2011)