SOFTWARE ENGINEERING

Spring 2024

RAPID SOFTWARE DEVELOPMENT

- Rapid development and delivery is now often the most important requirement for software systems
  - Businesses operate in a fast –changing requirement and it is practically impossible to produce a set of stable software requirements
  - Software has to evolve quickly to reflect changing business needs.
- Plan-driven development is essential for some types of system but does not meet these business needs.
- Agile development methods emerged in the late 1990s whose aim was to radically reduce the delivery time for working software systems

AGILE DEVELOPMENT

- Program specification, design and implementation are inter-leaved
- The system is developed as a series of versions or increments with stakeholders involved in version specification and evaluation
- Frequent delivery of new versions for evaluation
- Extensive tool support (e.g. automated testing tools) used to support development.
- Minimal documentation – focus on working code

PLAN-DRIVEN AND AGILE DEVELOPMENT

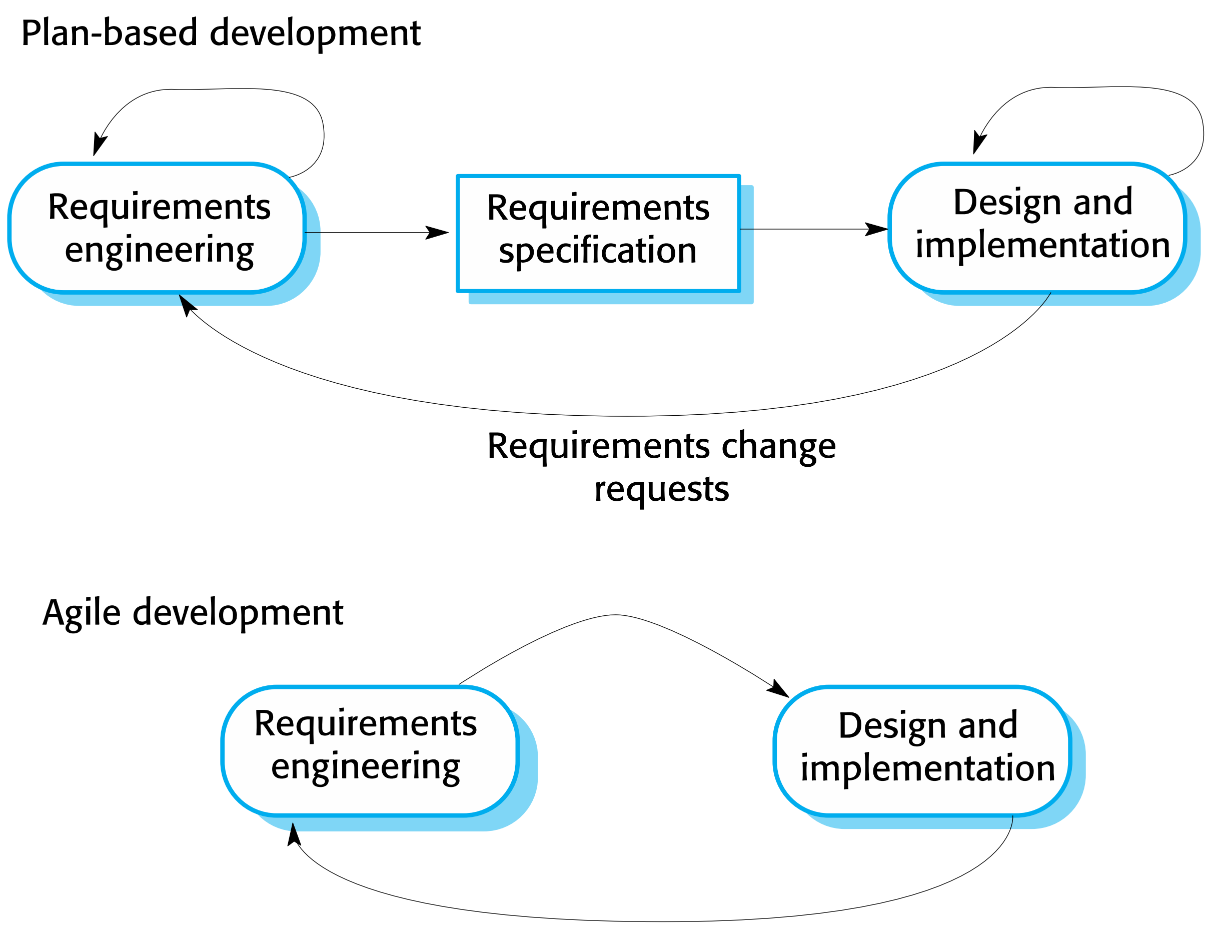
Plan-driven development

- A plan-driven approach to software engineering is based around separate development stages with the outputs to be produced at each of these stages planned in advance.
- Not necessarily waterfall model – plan-driven, incremental development is possible
- Iteration occurs within activities.

Agile development

- Specification, design, implementation and testing are inter-leaved and the outputs from the development process are decided through a process of negotiation during the software development process.

PLAN-DRIVEN AND AGILE DEVELOPMENT


Plan-based development
Requirements engineering
Requirements specification
Design and implementation
Requirements change requests
Agile development
Requirements engineering
Design and implementation

AGILE MANIFESTO

- *We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
  - *Individuals and interactions over processes and tools Working software over comprehensive documentation Customer collaboration over contract negotiation Responding to change over following a plan*

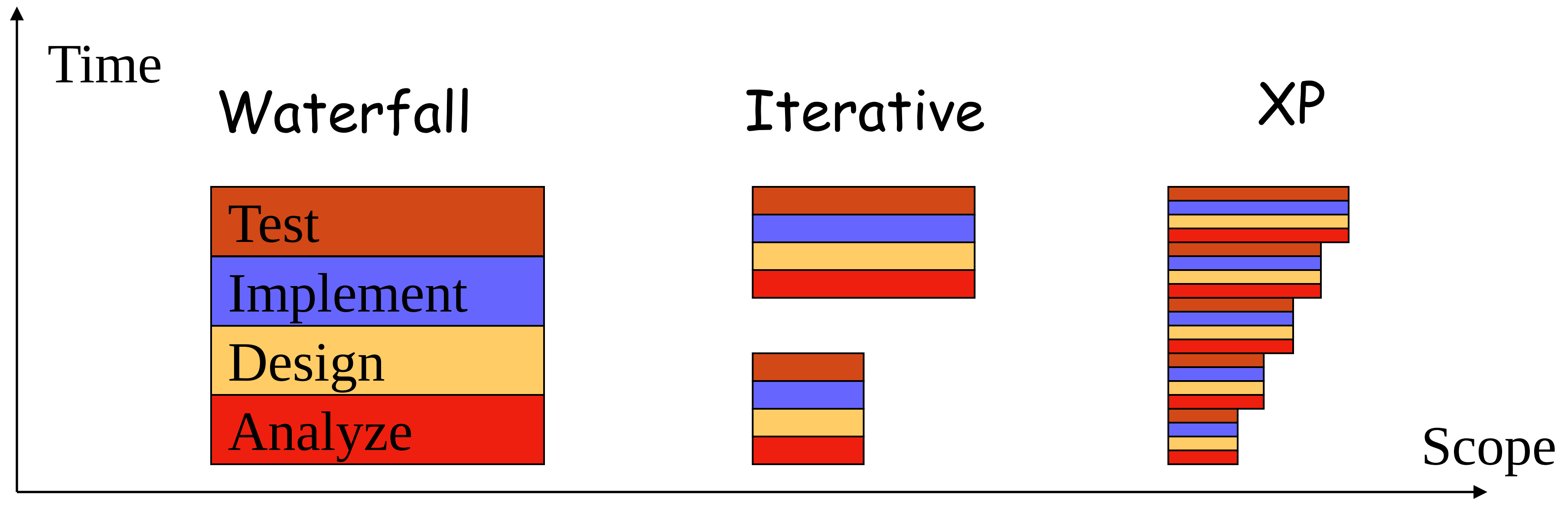
THE PRINCIPLES OF AGILE METHODS

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

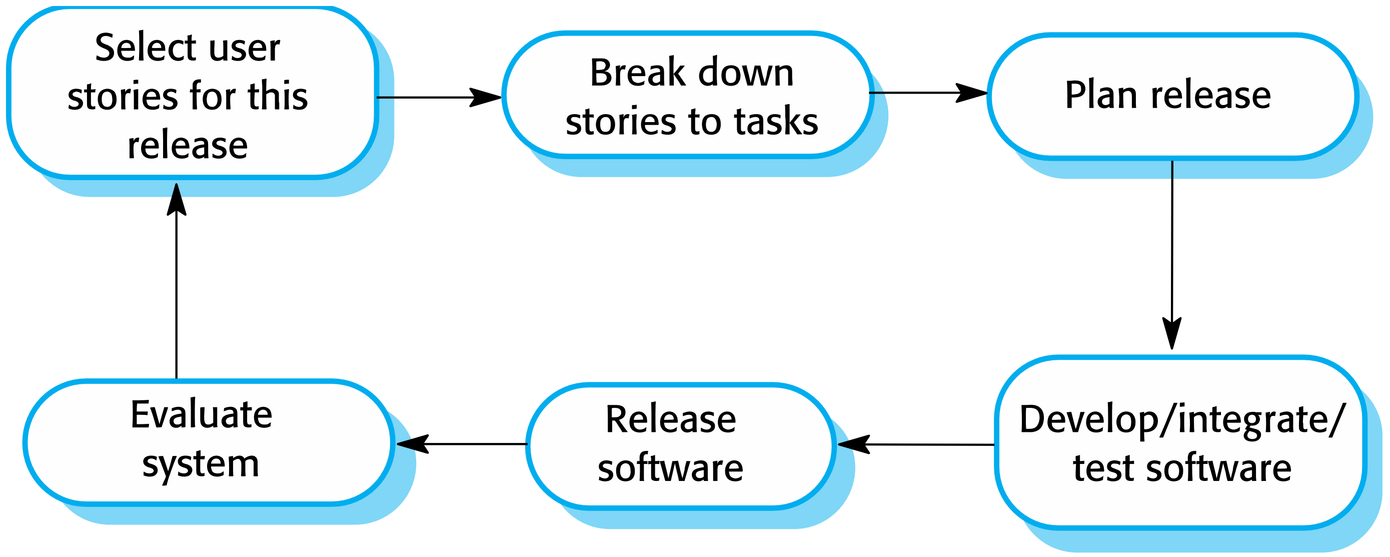
AGILE DEVELOPMENT TECHNIQUES

EXTREME PROGRAMMING (XP)

- XP: like iterative but taken to the *extreme*



THE EXTREME PROGRAMMING RELEASE CYCLE


Select user stories for this release
Break down stories to tasks
Plan release
Develop/integrate/ test software
Release software
Evaluate system

XP CUSTOMER

- Expert customer is part of the team
  - On site, available constantly
  - XP principles: communication and feedback
  - Make sure we build what the client wants

- Customer involved active in all stages:
  - Clarifies the requirements
  - Negotiates with the team what to do next
  - Writes and runs acceptance tests
  - Constantly evaluates intermediate versions
  - Question: How often is this feasible?

EXTREME PROGRAMMING PRACTICES (A)

Principle or practice	Description
Incremental planning	Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development 'Tasks'. See Figures 3.5 and 3.6.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

EXTREME PROGRAMMING PRACTICES (B)

Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

XP AND AGILE PRINCIPLES

- Incremental development is supported through small, frequent system releases.
- Customer involvement means full-time customer engagement with the team.
- People not process through pair programming, collective ownership and a process that avoids long working hours.
- Change supported through regular system releases.
- Maintaining simplicity through constant refactoring of code.

INFLUENTIAL XP PRACTICES

- Extreme programming has a technical focus and is not easy to integrate with management practice in most organizations.
- Consequently, while agile development uses practices from XP, the method as originally defined is not widely used.
- Key practices
  - User stories for specification
  - Refactoring
  - Test-first development
  - Pair programming

THE PLANNING GAME: USER STORIES

- Write on index cards (or on a wiki)
  - meaningful title
  - short (customer-centered) description

- Focus on "what" not the "why" or "how"

- Uses client language
  - Client must be able to test if a story is completed

- No need to have all stories in first iteration

EXAMPLE: ACCOUNTING SOFTWARE

- CEO: "I need an accounting software using which I can create a named account, list accounts, query the account balance, and delete an account."
- Analyze the CEO's statement and create some user stories

USER STORIES

Title: Create Account
Description: I can create a named account

Title: List Accounts
Description: I can get a list of all accounts.

Title: Query Account Balance
Description: I can query account balance.

Title: Delete Account
Description: I can delete a named account

USER STORIES

How is the list ordered?

Title: Create Account
Description: I can create a named account

Title: List Accounts
Description: I can get a list of all accounts.

Title: Query Account Balance
Description: I can query account balance.

Title: Delete Account
Description: I can delete a named account

USER STORIES

How is the list ordered?

Title: Create Account
Description: I can create a named account

Title: List Accounts
Description: I can get a list of all accounts. I can get an alphabetical list of all accounts.

Title: Query Account Balance
Description: I can query account balance.

Title: Delete Account
Description: I can delete a named account

USER STORIES

Title: Create Account

Description: I can create a named account

Title: List Accounts

Description: I can get a list of all accounts. I can ge[illegible] list of

Can I delete if a balance is not zero?

Title: Query Account Balance

Description: I can query account balance.

Title: Delete Account

Description: I can delete a named account

USER STORIES

Title: Create Account

Description: I can create a named account

Title: List Accounts

Description: I can get a list of all accounts. I can ge[illegible] list of

Can I delete if a balance is not zero?

Title: Query Account Balance

Description: I can query account balance.

Title: Delete Account

Description: I can delete a named account if the balance is zero.

USER STORY?

Title: Use AJAX for UI
Description: The user interface will use AJAX technologies to provide a cool and slick online experience.

USER STORY?

Title: Use AJAX for UI

Description: The user interface will use AJAX technologies to provide a cool and slick online experience.

Not a user story

CUSTOMER ACCEPTANCE TESTS

- Client must describe how the user stories will be tested
  - With concrete data examples,
  - Associated with (one or more) user stories

- Concrete expressions of user stories

USER STORIES

Title: Create Account
Description: I can create a named account

Title: List Accounts
Description: I can get a list of all accounts. I can get an alphabetical list of all accounts.

Title: Query Account Balance
Description: I can query account balance.

Title: Delete Account
Description: I can delete a named account if the balance is zero.

EXAMPLE: ACCOUNTING CUSTOMER TESTS

- Tests are associated with (one or more) stories

1. If I create an account “savings”, then another called “checking”, and I ask for the list of accounts I must obtain: “checking”, “savings”
2. If I now try to create “checking” again, I get an error
3. If now I query the balance of “checking”, I must get 0.
4. If I try to delete “stocks”, I get an error
5. If I delete “checking”, it should not appear in the new listing of accounts

…

AUTOMATE ACCEPTANCE TESTS

- Customer can write and later (re)run tests
  - E.g., customer writes an XML table with data examples, developers write tool to interpret table

- Tests should be automated
  - To ensure they are run after each release

TASKS

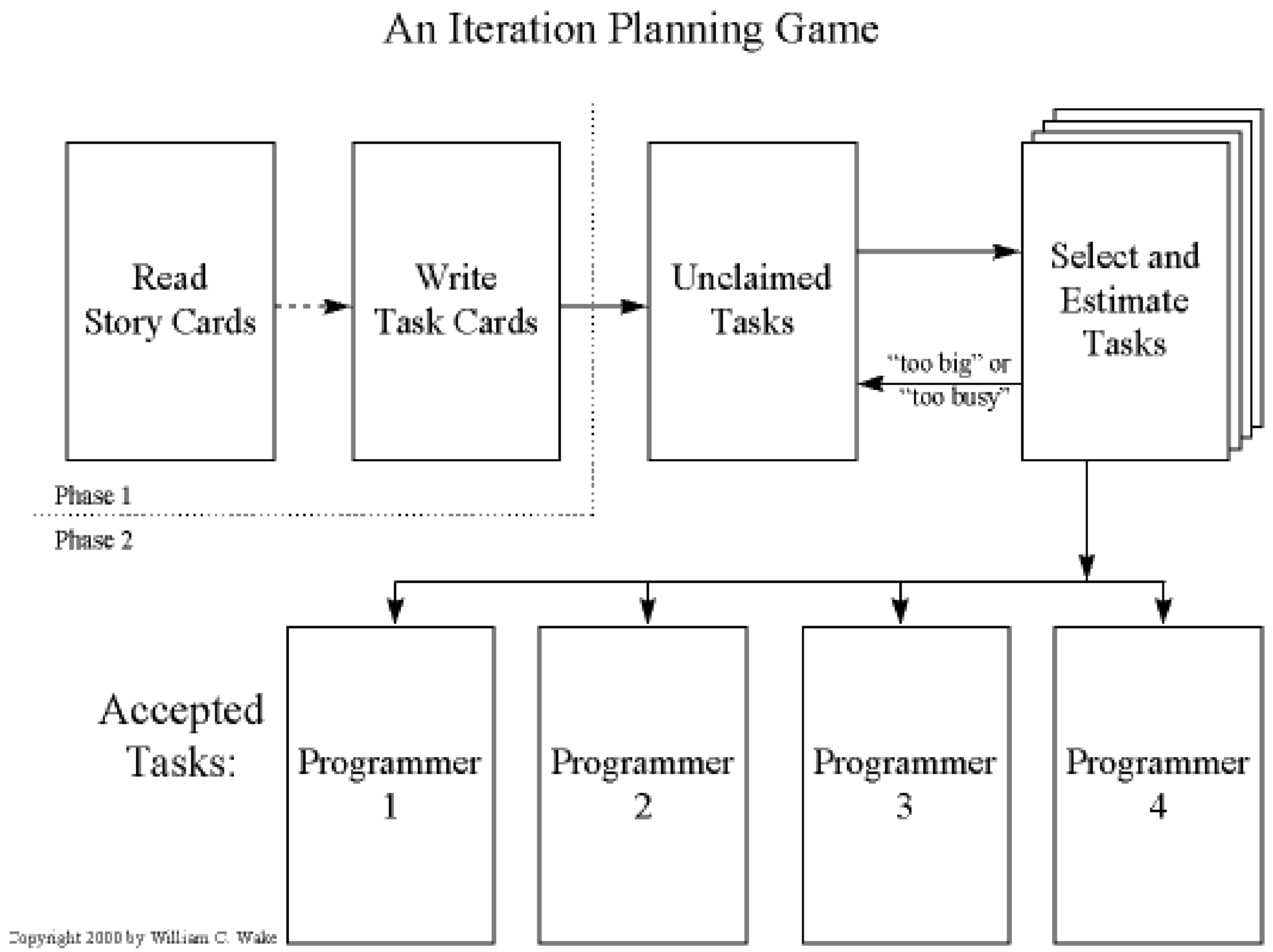
- Each story is broken into tasks
  - To split the work and to improve cost estimates
- Story: customer-centered description
- Task: developer-centered description
- Example:
  - Story: "I can create named accounts"
  - Tasks: "ask the user the name of the account"
    "check to see if the account already exists"
    "create an empty account"
- Break down only as much as needed to estimate cost
- Validate the breakdown of stories into tasks with the customer

TASKS

- If a story has too many tasks: break it down

- Team assigns cost to tasks
  - We care about relative cost of task/stories
  - Use abstract "units" (as opposed to hours, days)
  - Decide what is the smallest task, and assign it 1 unit
  - Experience will tell us how much a unit is
  - Developers can assign/estimate units by bidding: "I can do this task in 2 units"

PLAY THE PLANNING GAME



PLANNING GAME

- Customer chooses the important stories for the next release
- Development team bids on tasks
  - After first iteration, we know the speed (units/week) for each subteam
- Pick tasks => find completion date
- Pick completion date, pick stories until you fill the budget
- Customer might have to re-prioritize stories

TEST-DRIVEN DEVELOPMENT

- Write unit tests before implementing tasks
- Unit test: concentrate on one module
  - Start by breaking acceptance tests into units
- Example of a test

```
addAccount(“checking”);
if(balance(“checking”) != 0) throw …;
try { addAccount(“checking”);
      throw …;
} catch(DuplicateAccount e) { };
```

Think about names and calling conventions

Test both good and bad behavior

WHY WRITE TESTS FIRST?

- Testing-first clarifies the task at hand
  - Forces you to think in concrete terms
  - Helps identify and focus on corner cases
- Testing forces simplicity
  - Your only goal (now) is to pass the test
  - Fight premature optimization
- Tests act as useful documentation
  - Exposes (completely) the programmer's intent
- Testing increases confidence in the code
  - Courage to refactor code
  - Courage to change code

TEST-DRIVEN DEVELOPMENT. BUG FIXES

- Fail a unit test
  - Fix the code to pass the test
- Fail an acceptance test (user story)
  - Means that there aren't enough user tests
  - Add a user test, then fix the code to pass the test
- Fail on beta-testing
  - Make one or more unit tests from failing scenario
- Always write code to fix tests
  - Ensures that you will have a solid test suite

SIMPLICITY (KISS)

- Just-in-time design
  - design and implement what you know right now; don’t worry too much about future design decisions
- No premature optimization
  - You are not going to need it (YAGNI)
- In every big system there is a simple one waiting to get out

REFACTORING: IMPROVING THE DESIGN OF CODE

- Make the code easier to read/use/modify
  - Change "how" code does something
- Why?
  - Incremental feature extension might outgrow the initial design
  - Expected because of lack of extensive early design

REFACTORING: REMOVE DUPLICATED CODE

- Why? Easier to change, understand
- Inside a single method: move code outside conditionals

  if(…) { c1; c2 } else { c1; c3}

  c1; if(…) { c2 } else { c3 }
- In several methods: create new methods
- Almost duplicate code
  - … balance + 5 …   and … balance – x …
  - int incrBalance(int what) { return balance + what; }

    … incrBalance(5) …   and … incrBalance(- x) …

REFACTORING: CHANGE NAMES

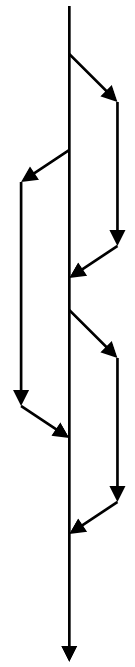
- Why?
  - A name should suggest what the method does and how it should be used
- Examples:
  - moveRightIfCan, moveRight, canMoveRight
- Meth1: rename the method, then fix compiler errors
  - Drawback: many edits until you can re-run tests
- Meth2: copy method with new name, make old one call the new one, slowly change references
  - Advantage: can run tests continuously

REFACTORING AND REGRESSION TESTING

- Comprehensive suite needed for fearless refactoring
- Only refactor working code
  - Do not refactor in the middle of implementing a feature
- Plan your refactoring to allow frequent regression tests
- Modern tools provide help with refactoring
- Recommended book: Martin Fowler's "Refactoring"

CONTINUOUS INTEGRATION

- Integrate your work after each task.
  - Start with official "release"
  - Once task is completed, integrate changes with current official release.
- All unit tests must run after integration
- Good tool support:
  - Hudson, CruiseControl



XP: PAIR PROGRAMMING

- Pilot and copilot metaphor
  - Or driver and navigator
- Pilot types, copilot monitors high-level issues
  - simplicity, integration with other components, assumptions being made implicitly
- Disagreements point early to design problems
- Pairs are shuffled periodically

PAIR PROGRAMMING

WE'RE GOING TO TRY SOMETHING CALLED EXTREME PROGRAMMING.
scottadams@aol.com
www.dilbert.com
FIRST, PICK A PARTNER. THE TWO OF YOU WILL WORK AT ONE COMPUTER FOR FORTY HOURS A WEEK.
© 2002 United Feature Syndicate, Inc.
1/9/03
THE NEW SYSTEM IS A MINUTE OLD AND I ALREADY HATE EVERYONE.

BENEFITS OF PAIR PROGRAMMING

- Results in better code
  - instant and complete and pleasant code review
  - copilot can think about big-picture
- Reduces risk
  - collective understanding of design/code
- Improves focus and productivity
  - instant source of advice
- Knowledge and skill migration
  - good habits spread

WHY SOME PROGRAMMERS RESIST PAIRING ?

- "Will slow me down"
  - Even the best hacker can learn something from even the lowliest programmer
- Afraid to show you are not a genius
  - Neither is your partner
  - Best way to learn

WHY SOME MANAGERS RESIST PAIRING?

- Myth: Inefficient use of personnel
  - That would be true if the most time consuming part of programming was typing !
  - 15% increase in dev. cost, and same decrease in bugs
- Resistance from developers
  - Ask them to experiment for a short time
  - Find people who want to pair

EVALUATION AND PLANNING

- Run acceptance tests
- Assess what was completed
  - How many stories ?
- Discuss problems that came up
  - Both technical and team issues
- Compute the speed of the team
- Re-estimate remaining user stories
- Plan with the client next iteration

XP PRACTICES

- On-site customer
- The Planning Game
- Small releases
- Testing
- Simple design
- Refactoring
- Metaphor
- Pair programming
- Collective ownership
- Continuous integration
- 40-hour week
- Coding standards

WHAT'S DIFFERENT ABOUT XP

- No specialized analysts, architects, programmers, testers, and integrators
  - every XP programmer participates in all of these critical activities every day.

- No complete up-front analysis and design
  - start with a quick analysis of the system
  - team continues to make analysis and design decisions throughout development.

WHAT'S DIFFERENT ABOUT XP

- Develop infrastructure and frameworks as you develop your application
  - not up-front
  - quickly delivering business value is the driver of XP projects.

WHEN TO (NOT) USE XP

- Use for:
  - A dynamic project done in small teams (2-10 people)
  - Projects with requirements prone to change
  - Have a customer available
- Do not use when:
  - Requirements are truly known and fixed
  - Cost of late changes is very high
  - Your customer is not available (e.g., space probe)

WHAT CAN GO WRONG?

- Requirements defined incrementally
  - Can lead to rework or scope creep

- Design is on the fly
  - Can lead to significant redesign

- Customer representative
  - Single point of failure
  - Frequent meetings can be costly

CONCLUSION: XP

- Extreme Programming is an incremental software process designed to cope with change

- With XP you never miss a deadline; you just deliver less content

AGILE SOFTWARE DEVELOPMENT

- “Agile Manifesto” 2001

“Scrum” project management

+ Extreme programming engineering practice

Build software incrementally, using short 1-4 week iterations

Keep development aligned with changing needs

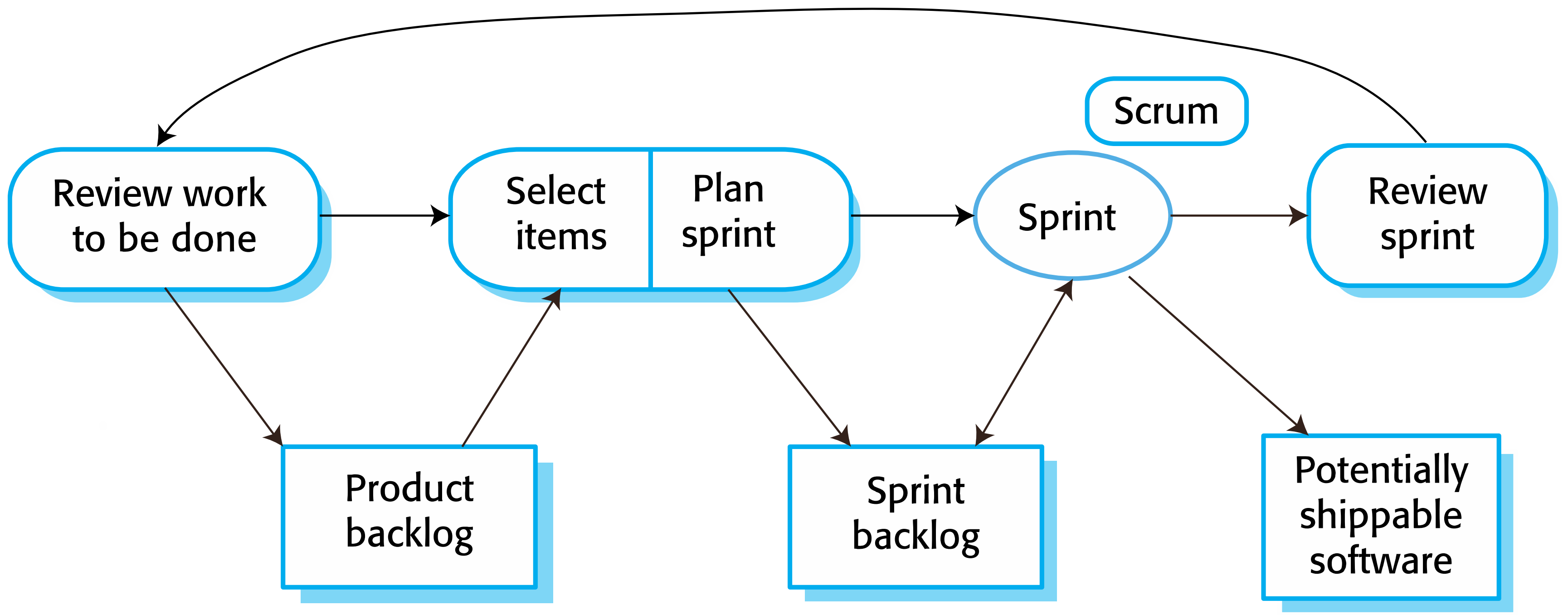
SCRUM TERMINOLOGY (A)

Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be ‘potentially shippable’ which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of ‘to do’ items which the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.

SCRUM TERMINOLOGY (B)

Scrum term	Definition
Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2-4 weeks long.
Velocity	An estimate of how much product backlog effort that a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

SCRUM SPRINT CYCLE



STRUCTURE OF AGILE TEAM

- Cross functional team
  - Developers, testers, product owner, scrum master

- Product Owner: Drive product from business perspective
  - Define and prioritize requirements
  - Determine release date and content
  - Lead iteration and release planning meetings
  - Accept/reject work of each iteration

STRUCTURE OF AGILE TEAM

- Cross functional team
  - Developers, testers, product owner, scrum master

- Scrum Master:Team leader who ensures team is fully productive
  - Enable close cooperation across roles
  - Remove blocks
  - Work with management to track progress
  - Lead the "inspect and adapt" processes

SCRUM

- "Process skeleton" which contains a set of practices and predefined roles
  - ScrumMaster (maintains processes)
  - Product Owner (represents the business)
  - Team (Designers/developers/testers)

- At each point:
  - User requirements go into prioritized backlog
  - Implementation done in iterations or sprints

SPRINT PLANNING

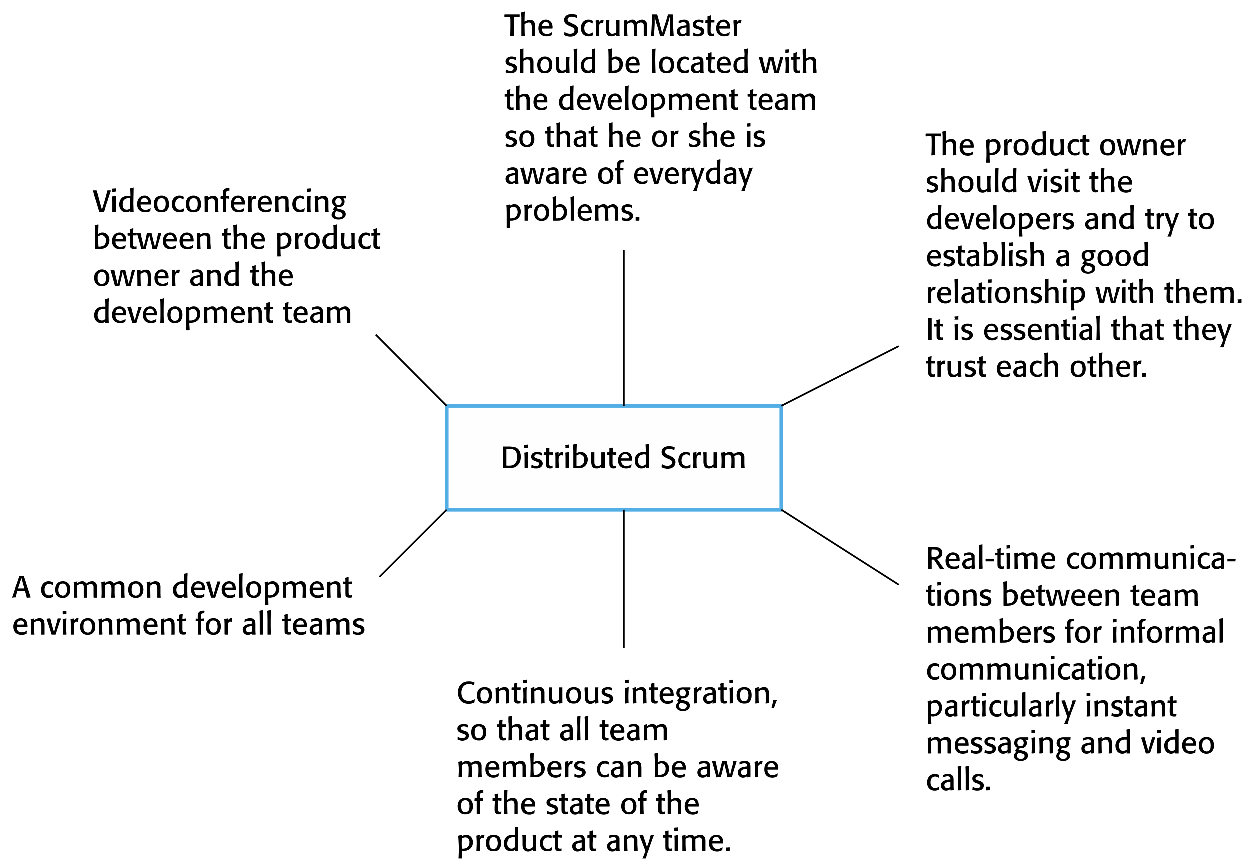
- Decide which user stories from the backlog go into the sprint (usually Product Owner)
- Team determines how much of this they can commit to complete
- During a sprint, the sprint backlog is frozen

MEETINGS: DAILY SCRUM

- Daily Scrum: Each day during the sprint, a project status meeting occurs

- Specific guidelines:
  - Start meeting on time
  - All are welcome, only committed members speak
  - Meeting lasts 15 min
- Questions:
  - What have you done since yesterday?
  - What are you planning to do today?
  - Do you have any problems preventing you from finishing your goals?

DISTRIBUTED SCRUM


Distributed Scrum
The ScrumMaster should be located with the development team so that he or she is aware of everyday problems.
Videoconferencing between the product owner and the development team
The product owner should visit the developers and try to establish a good relationship with them. It is essential that they trust each other.
A common development environment for all teams
Continuous integration, so that all team members can be aware of the state of the product at any time.
Real-time communica-tions between team members for informal communication, particularly instant messaging and video calls.

SCALING AGILE METHODS

SCALING AGILE METHODS

- Agile methods have proved to be successful for small and medium sized projects that can be developed by a small co-located team.
- It is sometimes argued that the success of these methods comes because of improved communications which is possible when everyone is working together.
- Scaling up agile methods involves changing these to cope with larger, longer projects where there are multiple development teams, perhaps working in different locations.

SCALING OUT AND SCALING UP

- ‘Scaling up’ is concerned with using agile methods for developing large software systems that cannot be developed by a small team.
- ‘Scaling out’ is concerned with how agile methods can be introduced across a large organization with many years of software development experience.
- When scaling agile methods it is importaant to maintain agile fundamentals:
  - Flexible planning, frequent system releases, continuous integration, test-driven development and good team communications.

PRACTICAL PROBLEMS WITH AGILE METHODS

- The informality of agile development is incompatible with the legal approach to contract definition that is commonly used in large companies.
- Agile methods are most appropriate for new software development rather than software maintenance. Yet the majority of software costs in large companies come from maintaining their existing software systems.
- Agile methods are designed for small co-located teams yet much software development now involves worldwide distributed teams.

CONTRACTUAL ISSUES

- Most software contracts for custom systems are based around a specification, which sets out what has to be implemented by the system developer for the system customer.
- However, this precludes interleaving specification and development as is the norm in agile development.
- A contract that pays for developer time rather than functionality is required.
  - However, this is seen as a high risk my many legal departments because what has to be delivered cannot be guaranteed.

AGILE METHODS AND SOFTWARE MAINTENANCE

- Most organizations spend more on maintaining existing software than they do on new software development. So, if agile methods are to be successful, they have to support maintenance as well as original development.
- Two key issues:
  - Are systems that are developed using an agile approach maintainable, given the emphasis in the development process of minimizing formal documentation?
  - Can agile methods be used effectively for evolving a system in response to customer change requests?
- Problems may arise if original development team cannot be maintained.

AGILE MAINTENANCE

- Key problems are:
  - Lack of product documentation
  - Keeping customers involved in the development process
  - Maintaining the continuity of the development team
- Agile development relies on the development team knowing and understanding what has to be done.
- For long-lifetime systems, this is a real problem as the original developers will not always work on the system.

AGILE AND PLAN-DRIVEN METHODS

- Most projects include elements of plan-driven and agile processes. Deciding on the balance depends on:
  - Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.
  - Is an incremental delivery strategy, where you deliver the software to customers and get rapid feedback from them, realistic? If so, consider using agile methods.
  - How large is the system that is being developed? Agile methods are most effective when the system can be developed with a small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.

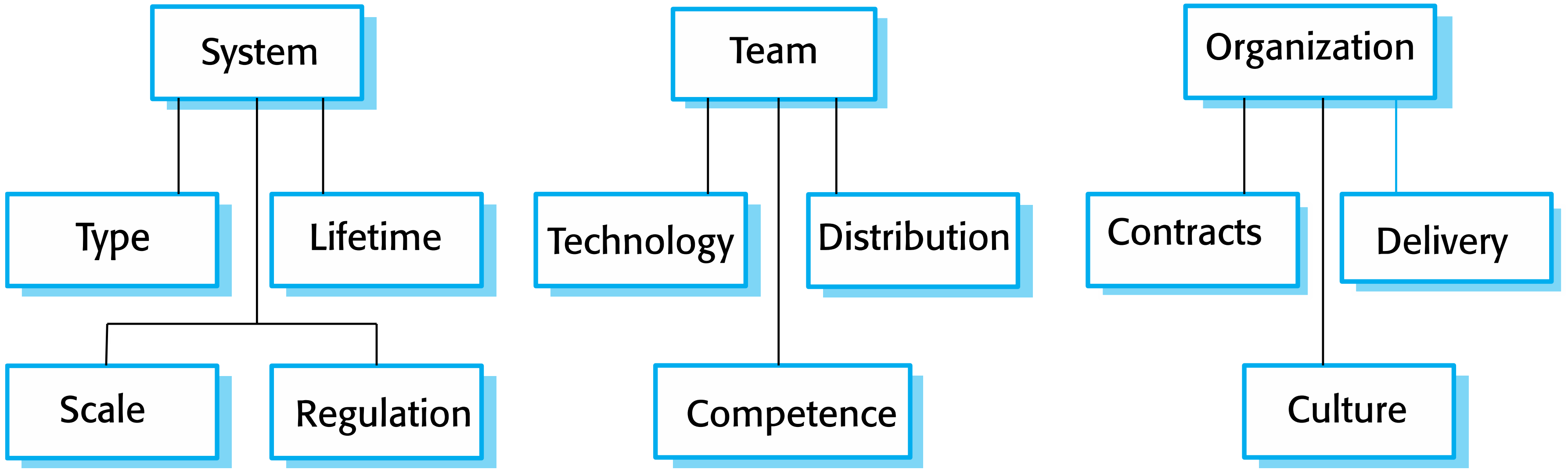
AGILE PRINCIPLES AND ORGANIZATIONAL PRACTICE

Principle	Practice
Customer involvement	This depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Often, customer representatives have other demands on their time and cannot play a full part in the software development. Where there are external stakeholders, such as regulators, it is difficult to represent their views to the agile team.
Embrace change	Prioritizing changes can be extremely difficult, especially in systems for which there are many stakeholders. Typically, each stakeholder gives different priorities to different changes.
Incremental delivery	Rapid iterations and short-term planning for development does not always fit in with the longer-term planning cycles of business planning and marketing. Marketing managers may need to know what product features several months in advance to prepare an effective marketing campaign.

AGILE PRINCIPLES AND ORGANIZATIONAL PRACTICE

Principle	Practice
Maintain simplicity	Under pressure from delivery schedules, team members may not have time to carry out desirable system simplifications.
People not process	Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods, and therefore may not interact well with other team members.

AGILE AND PLAN-BASED FACTORS



SYSTEM ISSUES

- How large is the system being developed?
  - Agile methods are most effective a relatively small co-located team who can communicate informally.
- What type of system is being developed?
  - Systems that require a lot of analysis before implementation need a fairly detailed design to carry out this analysis.
- What is the expected system lifetime?
  - Long-lifetime systems require documentation to communicate the intentions of the system developers to the support team.
- Is the system subject to external regulation?
  - If a system is regulated you will probably be required to produce detailed documentation as part of the system safety case.

PEOPLE AND TEAMS

- How good are the designers and programmers in the development team?
  - It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code.
- How is the development team organized?
  - Design documents may be required if the team is dsitributed.
- What support technologies are available?
  - IDE support for visualisation and program analysis is essential if design documentation is not available.

ORGANIZATIONAL ISSUES

- Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering.
- Is it standard organizational practice to develop a detailed system specification?
- Will customer representatives be available to provide feedback of system increments?
- Can informal agile development fit into the organizational culture of detailed documentation?

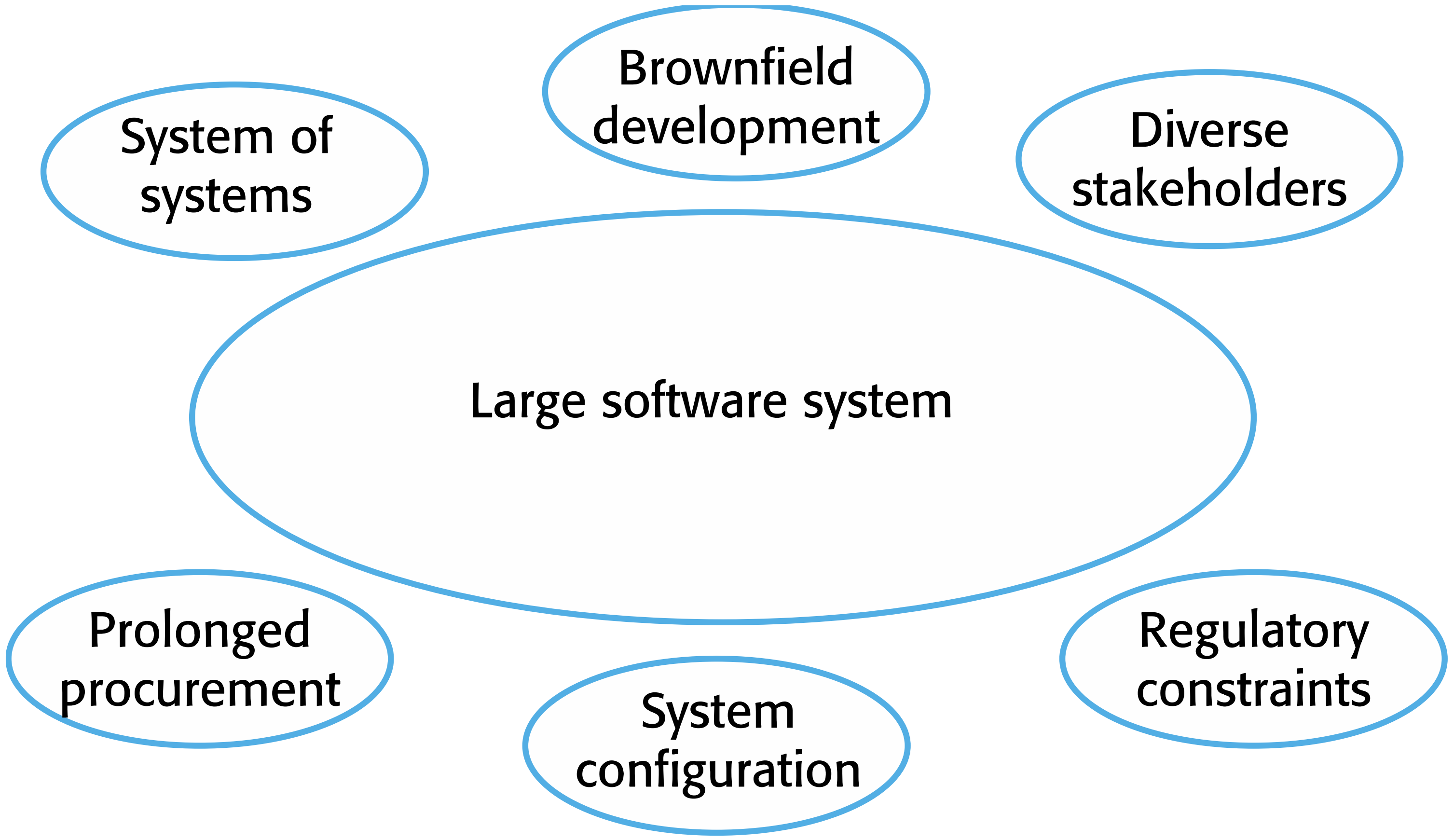
AGILE METHODS FOR LARGE SYSTEMS

- Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones.
- Large systems are 'brownfield systems', that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don't really lend themselves to flexibility and incremental development.
- Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development.

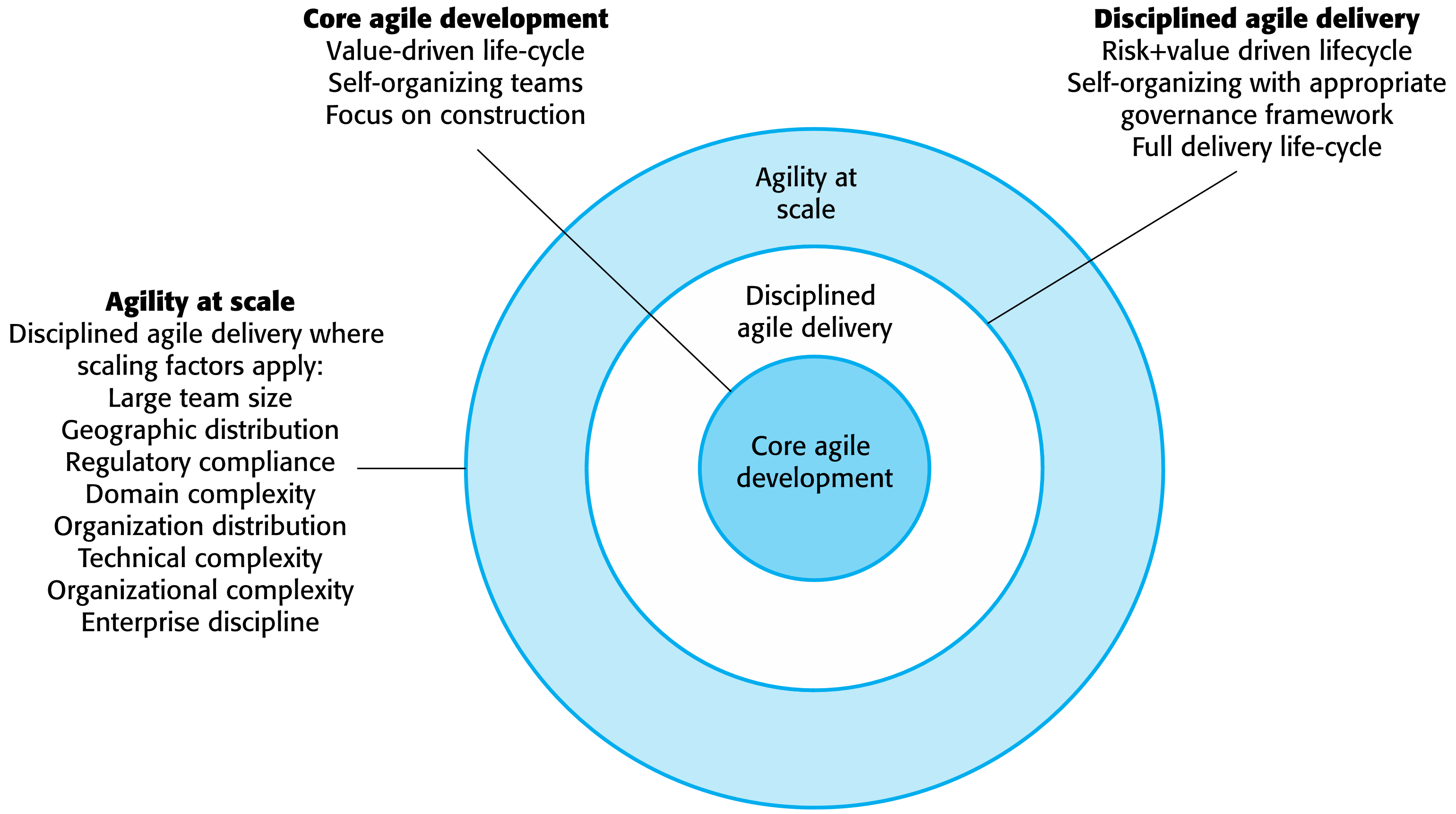
LARGE SYSTEM DEVELOPMENT

- Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed.
- Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.
- Large systems usually have a diverse set of stakeholders. It is practically impossible to involve all of these different stakeholders in the development process.

FACTORS IN LARGE SYSTEMS



IBM'S AGILITY AT SCALE MODEL


Core agile development
Value-driven life-cycle
Self-organizing teams
Focus on construction
Disciplined agile delivery
Risk+value driven lifecycle
Self-organizing with appropriate governance framework
Full delivery life-cycle
Agility at scale
Disciplined agile delivery
Agility at scale
Disciplined agile delivery where scaling factors apply:
Large team size
Geographic distribution
Regulatory compliance
Domain complexity
Organization distribution
Technical complexity
Organizational complexity
Enterprise discipline
Core agile development

SCALING UP TO LARGE SYSTEMS

- A completely incremental approach to requirements engineering is impossible.
- There cannot be a single product owner or customer representative.
- For large systems development, it is not possible to focus only on the code of the system.
- Cross-team communication mechanisms have to be designed and used.
- Continuous integration is practically impossible. However, it is essential to maintain frequent system builds and regular releases of the system.

MULTI-TEAM SCRUM

- *Role replication*
  - Each team has a Product Owner for their work component and ScrumMaster.
- *Product architects*
  - Each team chooses a product architect and these architects collaborate to design and evolve the overall system architecture.
- *Release alignment*
  - The dates of product releases from each team are aligned so that a demonstrable and complete system is produced.
- *Scrum of Scrums*
  - There is a daily Scrum of Scrums where representatives from each team meet to discuss progressand plan work to be done.

AGILE METHODS ACROSS ORGANIZATIONS

- Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach.
- Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods.
- Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities.
- There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.